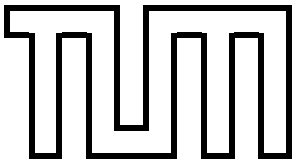


DEPARTMENT OF COMPUTER SCIENCE

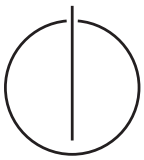
TECHNICAL UNIVERSITY OF MUNICH

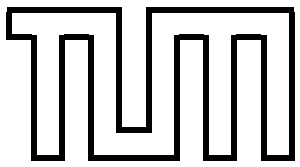
Master's Thesis in Computer Science

The Heavy Hitter Oracle: Enhancing Frequency Estimation in Skewed Data Streams

Lisa Schmierer

September 3, 2025





DEPARTMENT OF COMPUTER SCIENCE

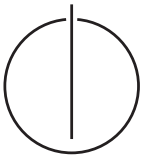
TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Computer Science

The Heavy Hitter Oracle: Enhancing Frequency Estimation in Skewed Data Streams

Das Heavy-Hitter-Orakel: Verbesserte Frequenzschätzung in verzerrten Datenströmen

Author:	Lisa Schmierer
Supervisor:	Prof. Dr. Thomas Neumann, Prof. Dr. Ioana-Oriana Bercea
Advisor:	Stefan Lehner, M.Sc.
Submission Date:	September 08, 2025



I confirm that this master's thesis in computer science is my own work and I have documented all sources and material used.

Munich, September 3, 2025 Lisa Schmierer

Acknowledgments

First but foremost, thanks to all the people who were directly involved in this thesis. To Stephan Lehner, Adrian Riedl, and Prof. Neumann: I am grateful for all your kind support. Thanks for always having an open ear, your helpful feedback, and support. It was a pleasure to work with you. Thanks for making working on this thesis across universities as smooth and uncomplicated as possible.

I genuinely enjoyed working on this thesis and that was mainly because of one person:
Dear Ioana, it was one of the greatest privileges I have ever enjoyed to work together with you and to be taught by you. Thanks for sparking my interest in algorithms – even for the pseudo-random ones – back in Advanced Algorithms. Thanks for kicking my ass when necessary, never allowing me to bullshit, and giving me prep-talks when I needed them most. Thanks for sharing your knowledge on the academic world and beyond (I will definitely check acknowledgements more often in the future). I learned way more from you than math, and how to feel comfortable writing on a whiteboard. I will miss not only working together with you but especially our regular breaks after long meetings and your unique way of making even tough problems fun. I sincerely hope that our paths will cross again.

Long before I started working on, or was even thinking about, this master's thesis, I had the pleasure of meeting people along the path that led to this point. In one of my favorite musical songs, the close friends Elphaba and Glinda from *Wicked* sing:

> *"I heard it say that people come into our lives for a reason*
> *Bringing something we must learn*
> *And we are led to those who help us most to grow*
> *If we let them and we help them in return*
> *I don't know if I believe that's true*
> *But I know I am who I am today because I knew you"*
>
> — For Good; Wicked

Thanks to all the people who shaped my path so far and left their handprint on my heart. Thanks for teaching me all the lessons – the hard and the fun ones.

Thanks, mum and dad, not least for your financial support. Lea and Daniel: I wouldn't wanna miss my weird Spotify algorithm. Thanks for holding down the fort at home.

I feel most fortunate and am grateful for calling some of the kindest, most fun, and open-hearted people I know my friends. Thanks for always having an open ear – even if the voice memo exceeds five minutes by far. Thanks for helping me to let sink in that it is not what

we achieve but who we are that makes us worth being loved. Thank you for being the most reliable support, whom I can always count on. I don't want to know how, especially the last two years, would have been without you. Unlike for Glinda and Elphaba, I hope there are many more memories ahead of us to make together.

There are more people that I connect a shaping memory with than I can possibly name on one (and a half – upsi) page(s). Thanks to all who believed in me long before I could. Thanks for encouraging and showing me how to dream. Thanks for reminding me to trust in the spark within me and not to get lost in the bright lights around me. People say that in hindsight, the dots connect. I sometimes wonder how much coincidence was involved that they connected exactly this way for me: Thanks to all who were there at what seems exactly the right time at the exact right place.

Beyond that, my sincere gratitude goes out to:

- all institutions that financially and ideally supported me. Without that support, my studies certainly would not have been possible in this form.
- Cher, Roy Bianco & Die Abbrunzati Boys, and Stephen Schwartz for delivering the suitable soundtrack during the last months.
- Brauerei Loscher and Alfred Nordquist for pushing me through the toughest times. I don't know how I would have survived some of the afternoon slumps and early mornings without club mate and coffee.
- Trish for never letting me down and being as reliable as a Swiss watch.

For many more adventures to come!

Abstract

This thesis presents new insights for frequency estimation on skewed data streams using predictions. The central task of frequency estimation is to approximate the frequencies of elements in a data stream when the number of distinct elements is too large to maintain exact counts for every item. The prediction-augmented setting assumes that an oracle is available that can predict the top-k elements, i.e., the k most frequent items in the stream.

Building upon CountSketch++, a novel algorithm for frequency estimation as suggested in [Aam+23], we argue by both theoretical analysis and practical experiments that it is too strong to assume a perfect oracle to predict the top-k elements for the prediction-augmented version of CountSketch++. We refer to the version of CountSketch++ that uses predictions as the learned version of the algorithm. To resolve this central shortcoming of learned CountSketch++, we propose and evaluate several realistic oracles for settings that allow one or two passes over the data stream.

To assess the power of these oracles, we compare their known theoretical bounds against the results from practical experiments on synthetic and real-world datasets. The experiments show that using the counter-based oracles Misra-Gries and SpaceSaving, one can recover the same weighted error as the learned version of the CountSketch++ that assumes a perfect oracle, even when using less space than required for their theoretical guarantees to hold. In fact, solely relying on the counts recovered by the SpaceSaving algorithm in a single pass as frequency estimates achieves the same performance in the experiments as the learned CountSketch++ algorithm that requires two passes over the data.

Additionally, we improve the theoretical analysis for the standard CountSketch++ algorithm, that is, the version without predictions, by a factor polynomial in $\log \log n$, where n is the number of distinct elements in the stream.

Contents

1. Introduction

Have you seen this item in the stream of items, and if so, how often? With unlimited resources, the solution to the problem is trivial: Maintain a counter for each distinct element and increase it whenever the item appears in the stream. For most real-world applications, however, the number of distinct elements, and hence the number of counters required for exact counts, exceeds the available memory by far. Additionally, the streams are so large and often unbounded that it is impossible to store and iterate over them multiple times, which would otherwise allow for stronger approximation algorithms.

This problem of frequency estimation with limited resources is one of the main challenges in data streaming. Closely related to it is finding *Heavy Hitters* or the top-k elements in a stream. *Heavy Hitters* describes all elements that occur more than a certain (relative) threshold in the stream, i.e., elements whose occurrences make up more than a certain proportion of the stream. Top-k elements, on the other hand, are the k most frequent elements in the stream, independent of any threshold.

These three problem settings – frequency estimation, Heavy Hitters, and top-k – occur in various real-world applications that need to process high-volume data. They include network traffic analysis, database and query optimization, natural language processing (NLP), sensor networks, and the Internet of Things (IoT). In network traffic analysis, observing frequent IP addresses helps to anticipate network congestion and detect malicious behavior like Distributed Denial of Service (DDoS) attacks [YNS16; Afe+16]. DDoS attacks, where adversaries flood the targeted network with an overwhelming number of requests to disrupt the service. For database systems, knowing frequent query terms allows programmers to optimize queries and resource allocation [RKA16]. In natural language processing, frequent word detection supports tasks such as keyword extraction and topic modeling [Zha+23]. In sensor networks and IoT, frequent data points can indicate anomalies or trends in environmental conditions, enabling timely responses to critical situations [HN+24].

Finding the exact solution for these problems in resource-constrained settings is impossible. However, there are algorithms that can approximate the underlying ground truth frequencies within mathematically proven bounds.

1.1. Motivation

CountMin [CM05] and CountSketches [CCFC04] are two common approaches to tackle the problem of frequency estimation. They are probabilistic data structures that use hash functions to map items to a fixed-sized array of counters, allowing for approximate frequency counts with low memory usage. While highly efficient, both approaches introduce errors due to hash collisions. Additionally, CountSketch and CountMin Sketches were designed independently of the data stream: Although they differ in how they aggregate estimates across the datastructure,

they both ignore underlying characteristics, such as the input and query distribution of the stream.

For most applications, these distributions are not arbitrary but follow known patterns. A common pattern is the power law, which describes the phenomenon that a small number of items accounts for a large proportion of the total frequency. This pattern is evident in network traffic, among other domains, where a few popular pages receive the majority of requests while many others are rarely accessed [New05]. A similar trend occurs in natural language processing, where a handful of common words, such as the articles "a" and "the", dominate usage, whereas most words rarely appear [LRA22]. These distribution patterns are either derived from historically learned data or their specific characteristics can be approximated directly from the input stream. Furthermore, some settings assume access to additional information about the input stream beyond its distribution, such as the identity of the most frequent items. Specifically, in these scenarios, an oracle that knows the top-k elements in the stream, i.e., the k most frequent items, is assumed to be available. While processing the data stream, the algorithm can query this oracle for advice to determine whether the current item is in the top-k or not. This knowledge allows for more efficient processing. For example, the algorithm can maintain exact counts for the most frequent items or allocate more space to approximate the frequency of Heavy Hitters while using less space to approximate smaller frequencies.

Existing approaches that leverage oracles to improve predictions have one common shortcoming: They make strong assumptions about the oracle and ignore the question of the actual implementation. Most often, existing approaches consider the oracle as a black box: They assume a (perfect) oracle is given and do not discuss how to realize it in practice. Other approaches assume a neural network to predict Heavy Hitters. While a deep-learning based approach is promising, it has several limitations: First, the theoretical analyzes for the algorithms do not include the space required to train and store the neural network in their memory bounds. Second, relying on a neural network as an oracle requires training the network separately for each domain. That again is a resource-intensive and time-consuming process that also depends on the availability of sufficient training data to create a reliable predictor.

In this thesis, we address these gaps as follows: First, we investigate how to further improve algorithms for frequency estimation by exploiting the underlying distribution of the data stream. We then examine in greater depth the limitations of existing approaches that rely on an oracle, and propose practical strategies for implementing such oracles. To do so, we build upon the CountSketch++ algorithm introduced in [Aam+23], a recent approach to improve frequency estimation with and without predictions, that is, knowing versus not knowing the top-k elements. As in [Aam+23], we focus on Zipfian distributions since they are predominant in many real-world applications [New05].

Our main contributions are:

- **Improved Bounds:** We prove a better upper bound for the CountSketch++ algorithm in the setting without predictions, as suggested in [Aam+23].
- **Theoretical Analysis and Improvements:** We provide new insights on how and why the CountSketch++ algorithm works and derive its shortcomings. This includes showing

that, given a perfect oracle, outputting zero for non-top-k elements achieves the same error as estimating their frequencies with the CountSketch++ algorithm.

- **Realistic Oracles:** We discuss how to implement realistic oracles to predict Heavy Hitters in a data stream in one and two passes over the data. For the one-pass setting, we investigate to what extent the oracles themselves can be used for frequency estimation and introduce augmented oracles that combine the oracle with an instance of CountSketch++.
- **Comprehensive Evaluation:** In addition to the theoretical findings, we comprehensively evaluate the approaches presented on three different datasets.

1.2. Related Work

There are two main types of algorithms for frequency estimation: random hashing-based algorithms, also referred to as sketches or sketch-based algorithms, and deterministic counter-based algorithms.

As the name suggests, hashing-based algorithms use hash functions to map items into fixed-size arrays of storing counters. They are highly efficient but introduce errors due to hash collisions. One of the most prominent sketch-based algorithms is the CountSketch, also referred to as CountMedian Sketch, as introduced in [CCFC04]. The main idea is to use multiple hash functions to map items into an array and output the median over all corresponding counters to estimate item frequencies. Three years after the invention of CountSketches, [CM05] suggested the CountMin Sketches. Similar to CountSketches, CountMin makes use of multiple hash functions. However, instead of taking the median, the algorithm outputs the minimum value over all hash rows for each item. CountMin and CountSketch also differ in how they aggregate item counts in the array. We will provide further details on the algorithms in Chapter 2.

Deterministic counter-based algorithms, on the other hand, maintain counters for a fixed number of items. The results are deterministic and, different from sketch-based algorithms, may depend on the order of the items in the stream. The best-known counter-based algorithms include Misra-Gries, as suggested in [MG82], and SpaceSaving introduced by [MAEA05a]. Given a fixed number of counters, these algorithms guarantee to find all elements that account for more than a certain proportion of the total stream. Initially, most counter-based algorithms were designed to find the top-k elements. However, they also provide counts for the predicted top elements. These can be used for frequency estimation as well. As for the sketch-based algorithms, we will provide further details on the counter-based algorithms in Chapter 2. Additionally, [CH08] provides a comprehensive overview and comparison of the algorithms mentioned above.

Recent works have explored enhancements and adaptations for specific use cases based on these foundational algorithms. [Hsu+19] introduced a framework for learning-based frequency estimation that combines machine learning methods with classic algorithms. In their work, they rely on a trained neural network as an oracle that can differentiate between heavy and non-heavy elements. Based on these predictions, the mechanism stores heavy elements separately and processes the remaining elements with a sketching algorithm of choice. [MV22] provides an overview that outlines the power of predictions for frequency estimation and

Heavy Hitter detection. [SM24] expands the idea of learning-based frequency estimation to the class of counter-based algorithms, more specifically, to the SpaceSaving algorithms.

Following the approach [CCFC04] suggested for CountSketch, [RKA16], combines a heap for frequent items with a CountMin Sketch. However, in [RKA16], the authors focus primarily on how keeping a heap improves run-time when querying the data structure. Instead of relying on a heap, [MAEA05b] improves estimations by using different numbers of hash functions depending on whether an external oracle labels the item as a Heavy Hitter or not.

[Aam+25] generalizes learning-based frequency estimation to high-dimensional settings, where the goal is to find frequent directions. They use a learning-based Misra-Gries algorithm that uses parts of the available space to store exact counts for the predicted top-k elements. The rest of the space is used to run a version of the Misra-Gries algorithm for the remaining elements.

Focusing on Zipfian distributions, [Aam+23] proves (nearly) tight bounds for CountMin and CountSketch and suggests a new algorithm, that works with and without predictions. Following [Aam+25], we will refer to this algorithm as CountSketch++ (without predictions) and learned CountSketch++ (with predictions) respectively. The approach combines multiple CountSketches to improve the weighted error of the estimates. Chapter 3 in this thesis contains a detailed description of the algorithm, as we use the algorithm as a starting point for our work.

1.3. Overview

Following this section, **Chapter 2** provides the necessary theoretical and technical foundations for this work. It covers different data streaming models, the error definition we will use throughout the work, and introduces the most established algorithms for frequency estimation, Heavy Hitters, and top-k detection.

Chapter 3 comprises the main contributions of the thesis. It introduces the CountSketch++ algorithm developed in [Aam+23], on which we build our work. We will present improved theoretical bounds, discuss shortcomings of the algorithm, and present solutions for these. This also includes the question of how to implement oracles in practice.

Chapter 4 presents experiments and practical results for the theoretical findings stated in Chapter 3. Finally, **Chapter 5** outlines potential directions for future research, before summarizing the thesis in **Chapter 6**.

2. Preliminaries

In addition to the notation we will use throughout the thesis, this chapter defines the exact problem setup. This includes assumptions about the data stream and the error definition. It also covers the most relevant algorithms for frequency estimation, Heavy Hitter and top-k estimation, which we will build upon in the following chapters.

2.1. Problem Setup

Not all the algorithms used throughout this paper can handle deletions. To make them comparable, we therefore assume data streams only contain insertions (*Cash Register Model*). Each update is of the form (i, Δ), where i is the element and Δ is the frequency increment. Whether we limit the frequency increment Δ to one or not does not matter for our purposes as we can rewrite any stream of the form (i, Δ) to a stream of the form $(i, 1)$ by inserting i Δ-times after each other.

2.1.1. Notation

$\lesssim$ denotes inequality up to a constant factor, i.e., $\exists C > 0 : x \lesssim y \Leftrightarrow x \leq C \cdot y$. To simplify notation, we describe sets in square brackets, e.g., $[n] = \{1, 2, \ldots, n\}$, and assume that $f_1 \geq f_2 \geq f_3 \geq \cdots \geq f_n$, where f_i is the frequency of element i. Space complexity is measured in words of memory.

Notation	
B	Available space budget in words of memory
N	Length of the data stream
n	Number of distinct elements in the data stream
k	Number of counters in counter-based algorithms. Equivalent to the number of top elements in top-k algorithms.
i	An element in the data stream
Δ	Frequency increment upon arrival of element i
f_i	True frequency of element i
$\hat{f}_i$	(Final) Estimated frequency of element i
$\tilde{f}_i$	Estimate obtained in the filtering step of CountSketch++. I.e., median over medians obtained from small sketch tables.
$\hat{f}_i^{(k)}$	Estimated frequency of element i in CountSketch++ given by the k^{th} small CountSketch table

A	Scaling factor for Zipfian frequencies, $A \sim \frac{N}{\log n}$ for $\alpha = 1$
d, w	Depth (number of rows) and width (number of columns) of a sketch table (e.g., CountMin or CountSketch)
$h_1, h_2 \dots h_d$	Hash functions that map the universe of elements $[n]$ to the buckets $[0, w-1]$ in sketch-based algorithms
$s_1, s_2, \dots s_d$	Sign hash functions that map the universe of elements $[n]$ to either -1 or $+1$ in CountSketch
$C_j[h_j(i)]$	Bucket in row j, in which hash function h_j maps element i
C_j	Value of counter at position j in a heap
C_{min}	Minimum value among all counters in a heap
$C_{\text{diff}}[i]$	Difference of counts since last insertion of element i into the heap of exact counters in sketch-based oracles
T	Number of small CountSketch tables in (learned) CountSketch++, $T = \log \log n$
τ	Predefined threshold for filtering rare elements in CountSketch++, $\tau = \frac{A \cdot \mathcal{O}(\log \log n)}{B}$

2.1.2. Zipfian Distribution

Building on [Aam+23], we assume that frequencies follow a Zipfian distribution throughout this work, as many real-world datasets in relevant domains exhibit heavy-tailed distributions [New05]. For instance, in social networks, a small number of users have disproportionately many followers, while the majority have only a few. Likewise, in web traffic, a handful of highly popular websites attract the majority of visits, whereas most receive relatively little hits. Understanding and exploiting these properties is essential for accurate frequency estimation and effective analysis of real-world data.
In a Zipfian distribution, the frequency of an element is inversely proportional to its rank in the frequency table. I.e., the i^{th} largest frequency f_i is proportional to A/i^{α}, for some A and α, where A is a scaling factor and α controls the skewness of the distribution. Since the total stream length N is equal to the sum over all frequencies, it holds for A that:

$$N = \sum_{i=1}^{n} f_i = \sum_{i=1}^{n} \frac{A}{i^{\alpha}} \Rightarrow A = \frac{N}{\sum_{i=1}^{n} \frac{1}{i^{\alpha}}}$$

Throughout the paper, we assume $\alpha = 1$ as in [Aam+23]. For all $\alpha > 1$ the distribution is even more skewed, which should make the problem easier, especially for finding top-k elements. In fact, [Ber+10] have proven that this is indeed the case for the SpaceSaving algorithm. We expect this effect to apply to our setting as well. Putting everything together:

$$f_i \propto \frac{1}{i}$$

2.1.3. Error Model

For our analysis, we adopt the weighted error model from [Aam+23] and [Hsu+19]. The authors in [Hsu+19] argue that the simple weighted objective is a more comprehensive measure, since it does not require tuning two separate parameters as in the traditional (ϵ, δ)-error formulation, where ϵ describes the (expected) error and δ the probability, that the actual error exceeds this value. In their view, this makes it a more natural choice from an ML standpoint and therefore more suitable for learned frequency estimation. The weighted error describes the expected error under the weighted distribution.

$$\textit{Weighted Error} = \frac{1}{N} \sum_{i \in [n]} f_i \cdot |\hat{f}_i - f_i|$$

One way of interpreting the error above is that it corresponds to the expected error we get if we assume that queries are drawn from the same distribution as the item frequencies. I.e., the more frequent an element occurs in the stream, the more likely it is queried. Therefore, weighting the error by the true frequency reflects how important elements are for the querying phase. Weighting the error that way introduces a bias towards heavy elements. It favours algorithms with a lower error for heavy elements over algorithms with a higher error for these elements, even if the total sum of absolute deviations, i.e., the unweighted estimation error, is the same. It might even lead to situations in which an algorithm A that outperforms an algorithm B in terms of the unweighted error, performs worse when looking at the weighted error, and vice versa.

2.2. Frequency Estimation

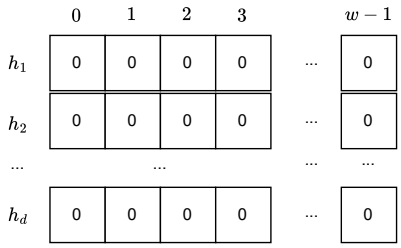
The goal of frequency estimation is to find a good approximate $\hat{f}_i$ for the true frequency f_i of each element i in a data stream with limited resources. Frequency estimation algorithms are especially useful when dealing with massive data where exact counting is infeasible. There are two main types: sketch-based algorithms and counter-based algorithms. In this section, we will focus on sketch-based algorithms for frequency estimation and discuss two popular methods: CountMin and CountSketch. Many frequency estimation algorithms can be adjusted for Heavy Hitter / top-k estimation and vice versa, which we will discuss in sections 2.2.3 and 2.3.3 respectively.

2.2.1. CountMin Sketch

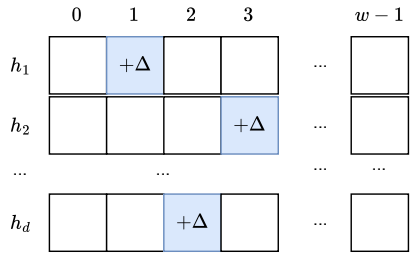
CountMin is a probabilistic data structure for frequency estimation introduced by Cormode and Muthukrishnan in [CM05]. It consists of a two-dimensional array with space parameters d and w, where d is the depth (number of rows/hash functions) and w is the width (number of hash buckets per row), resulting in a total space $B = w \cdot d$.

The data structure uses d independent hash functions $h_1, h_2, \ldots, h_d$, each mapping elements to bucket indices in the range $[0, w-1]$.

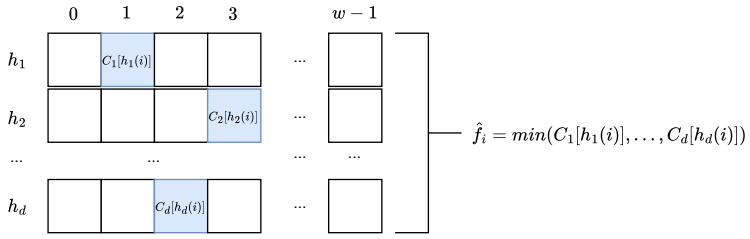
Initialization To initialize the CountMin Sketch, the algorithm creates a two-dimensional array of size $d \times w$, where all counters are initialized to zero (cf. Figure 2.1a).



(a) Initialization of a CountMin Sketch with d hash functions and width w.



(b) Update operation of a CountMin Sketch for element i with frequency increment Δ.



(c) Query operation of a CountMin Sketch for element i.

Figure 2.1.: Initialization, update, and query operation for a CountMin Sketch.

Update Operation When an element i arrives with frequency increment Δ, the CountMin Sketch updates its internal state as shown in Figure 2.1b: For each hash function h_j, the algorithm determines the bucket to which the hash function j maps element i, $h_j(i)$, and increments the corresponding counter by Δ: $C_j[h_j(i)] \mathrel{+}= \Delta$

Query Operation To estimate the frequency of an element i, the CountMin Sketch first retrieves all counts from each respective hash function (cf. Figure 2.1c). It then returns the minimum value among all these counters:

$$\hat{f}_i = min_{j \in [1,d]} C_j[h_j(i)]$$

Guarantees The CountMin Sketch provides a one-sided error guarantee: The estimated frequency $\hat{f}_i$ is always greater than or equal to the true frequency f_i. It, therefore, never underestimates the true count. For Zipfian distributions, [Aam+23] proves tight bounds for the weighted error. Given B words of memory, the weighted error is:

$$\Theta\left(\frac{N}{B}\right)$$

In the same paper, the authors also show a tight bound for a setting with predictions, which they call *learned CountMin*. This setup knows the top-k elements and tracks them in a separate heap outside the sketch, while the CountMin Sketch handles the remaining elements. In this

scenario, the weighted error for Zipfian data and given B words of memory is:

$$\Theta\left(\frac{N}{\log(n)} \cdot \frac{\log^2\left(\frac{n}{B}\right)}{B\log(n)}\right)$$

Additionally, [Aam+23] shows that a constant number of hash functions is asymptotically optimal.

2.2.2. CountSketch

Like CountMin, CountSketch is a probabilistic data structure designed to estimate the frequencies of elements in a data stream using sublinear space. First introduced by Charikar et al. in [CCFC04], CountSketch consists of a two-dimensional array with space parameters d and w, where d is the depth (number of rows / hash functions) and w is the width (number of buckets per row), resulting in a total space $B = w \cdot d$.

Each row uses independent hash functions, providing d different estimates for any given element. To do so, the data structure employs two types of (pairwise independent) hash functions:

1. **Bucket hash functions** $h_1, h_2, \ldots, h_d$: maps an element to a bucket index in the range $[0, w-1]$

2. **Sign hash functions** $s_1, s_2, \ldots, s_d$: maps an element to either $+1$ or -1

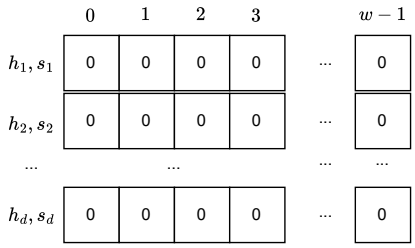
Initialization To initialize the CountSketch, the algorithm creates a two-dimensional array of size $d \times w$, where all counters are initialized to zero (cf. Figure 2.2a).

Like CountMin, CountSketch supports two primary operations: update and query.

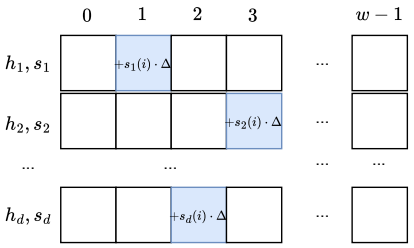
Update Operation When an element i arrives with frequency increment Δ, CountSketch updates its internal state as illustrated in Figure 2.2b: For each hash function h_j, the algorithm determines the bucket to which the hash function maps element i. Additionally, it computes the sign ($+1$ or -1) for the element and afterwards updates the corresponding counter by Δ multiplied by the result of the sign-hash function, $s_j(i)$: $C_j[h_j(i)] += s_j(i) \cdot \Delta$

Query Operation To estimate the frequency of an element i, CountSketch, like CountMin, first collects all counts corresponding to the hash rows. It then returns the median of the d different estimates (cf. Figure 2.2c):

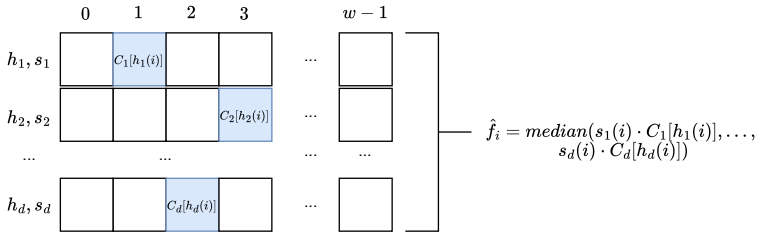
$$\hat{f}_i = median_{j \in [1,d]} s_j(i) \cdot C_j[h_j(i)]$$



(a) Initialization of a CountSketch with d hash functions and width w



(b) Update operation of a CountSketch for element i with frequency increment Δ



(c) Query operation of a CountSketch for element i

Figure 2.2.: Initialization, update and query operation for a CountSketch.

Guarantees For Zipfian distributions, [Aam+23] prove tight bounds for the weighted error. Given B words of memory, the weighted error is:

$$\Theta\left(\frac{N}{B\log(n)}\right)$$

Like for CountMin, the authors also show a tight bound for a version with predictions: This setup knows the top-k elements and tracks them in a separate heap outside the sketch. The CountSketch only tracks the remaining elements. In this scenario, the weighted error for Zipfian data and B words of memory is:

$$\Theta\left(\frac{N}{\log(n)}\frac{\log\left(\frac{n}{B}\right)}{B\log(n)}\right)$$

Matching the result for CountMin, [Aam+23] shows that for a CountSketch, a constant number of hash functions is asymptotically optimal.

2.2.3. Counter-Based Algorithms

Although many counter-based algorithms are tailored to solve the Heavy Hitters / top-k problem, they work for frequency estimation as well. The simplest way to adapt them is to run the algorithms as given and return the final counter values for tracked elements and estimate a count of zero for all others.

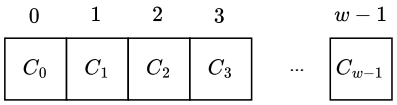


Figure 2.3.: To use counter-based algorithms for frequency estimation, output the corresponding counter values for elements that are tracked and zero otherwise.

2.3. Heavy Hitter and Top-k Estimation

Heavy Hitters algorithms aim to identify elements in a data stream whose frequency exceeds a given proportion/threshold of the total stream length. Essentially, they find those elements that stand out due to their high relative occurrence. Top-k algorithms, on the other hand, focus on retrieving the k most frequent elements, regardless of how much more frequent these elements are compared to others. Most algorithms, however, can be adjusted to solve both problems. In this work, we will focus on the Misra-Gries and the SpaceSaving algorithm for the Heavy Hitters problem, as well as heap-based sketch algorithms for top-k estimation.

2.3.1. Misra-Gries Algorithm

The Misra-Gries algorithm is a counter-based approach for finding Heavy Hitters in a data stream. It is deterministic, i.e., if the order of elements stays the same, the algorithm will always return the same Heavy Hitters. The algorithm maintains a fixed number of counters for the most frequent elements and updates these counters as new elements arrive as described in Algorithm 1. If an element is already being tracked, it increments the corresponding counter each time the element appears in the stream. If the element is not being tracked yet, and there is still unoccupied space left, the element is added to the list of counters with a counter initialized to one. If the list of counters is full, the algorithm decrements all counters by one. If a counter reaches zero, the algorithm removes the corresponding key/element from the list of counters and frees up the space to track other elements.

Algorithm 1 Misra-Gries Algorithm

Require: Stream of elements $1, 2, \ldots, n$; space parameter k
Initialize empty dictionary C (counters)
for each element i in the stream **do**
    if $i \in C.keys()$ **then**
        $C[i] \leftarrow C[i] + 1$
    else if $|C| <= k - 1$ **then**
        $C[x] \leftarrow 1$
    else
        for each key $y \in C.keys()$ **do**
            $C[y] \leftarrow C[y] - 1$
            if $C[y] = 0$ **then**
                Remove y from C
return C

Guarantees Let N be the stream length, f_i be the true frequency of an element i in the stream, and $\hat{f}_i$ be the estimated frequency of element i as approximated by the Misra-Gries algorithm. The following lemma holds:

Misra-Gries Guarantee *Source: [MG82]*

(1) For any $k > 0$, Misra-Gries with k counters can output $\hat{f}_i$ such that

$$f_i - \frac{N}{k+1} \leq \hat{f}_i \leq f_i.$$

(2) If an element i appears more than $\frac{N}{k+1}$ times in the stream, it is guaranteed to be in the output of the algorithm.

In other words, the algorithm never underestimates the frequency of an element by more than $\frac{N}{k+1}$, and never overestimates it. Additionally, the algorithm is guaranteed to find all elements that appear strictly more than $\frac{N}{k+1}$ times in the stream. If fewer than k elements in the stream satisfy this condition, the algorithm may return additional elements arbitrarily. Which ones these are depends on the order of elements in the stream. While the Misra-Gries algorithm is primarily designed to identify frequent items (Heavy Hitters), it can also be used for frequency estimation by returning the current counter values for tracked elements and assuming a count of zero for all others.

2.3.2. SpaceSaving Algorithm

Like the Misra-Gries algorithm, SpaceSaving is a deterministic algorithm that uses counters to track the most frequent elements. It uses the same data structure as the Misra-Gries algorithm, i.e., a dictionary of counters, but differs in the update mechanism. Contrary to the Misra-Gries algorithm, SpaceSaving always adds an element to the list of counters. It executes the steps as described in Algorithm 2: Whenever a new element arrives, the algorithm checks if it is already being tracked. If so, it increases the corresponding counter by one. Otherwise, it checks if there is still unoccupied space in the list of counters. If so, it adds the new element with a counter initialized to one. So far, this is the same as in the Misra-Gries algorithm. What differs is the case when the list of counters is full: If the element is not part of the heap already and the counter heap is fully occupied, SpaceSaving increments the smallest counter and replaces the corresponding key with the new element. That way, SpaceSaving never underestimates the true frequency of an element, but may overestimate it.

Algorithm 2 SpaceSaving Algorithm

Require: Stream of elements $1, 2, \ldots, n$, space parameter k

```
Initialize empty dictionary C of size at most k
for each element i in the stream do
    if i ∈ C.keys() then
        C[i] ← C[i] + 1
    else if |C| < k then
        C[i] ← 1
    else
        Let y be the key in C associated with the minimum count
        C[i] ← C[y] + 1
        Remove y from C
return C
```

Guarantees Let N be the stream length, f_i be the true frequency of an element i in the stream, and $\hat{f}_i$ be the estimated frequency of i by the SpaceSaving algorithm. The following lemma holds:

SpaceSaving Guarantees *Source: [MAEA05a]*

(1) For any $k > 0$, SpaceSaving with k counters can output $\hat{f}_i$ such that

$$f_i \leq \hat{f}_i \leq f_i + \frac{N}{k}.$$

(2) If an element i appears more than $\frac{N}{k}$ times in the stream, it is guaranteed to be in the output of the algorithm.

In other words, the algorithm never overestimates an element's frequency by more than $\frac{N}{k}$, and never underestimates it. Additionally, the algorithm is guaranteed to find all elements that appear strictly more than $\frac{N}{k}$ times in the stream. Equivalently to the Misra-Gries algorithm, it holds that if fewer than k elements in the stream satisfy this condition, the algorithm may return additional elements arbitrarily. In the same way as Misra-Gries, SpaceSaving can be used for frequency estimation too.

2.3.3. Heap-Based Sketches

There are two main differences between heap-based sketches and the two previous algorithms: First, both Misra-Gries and SpaceSaving are deterministic, whereas heap-based sketches rely on universal hashing, which introduces randomness by selecting a hash function at random from a family that satisfies specific mathematical properties. Second – as mentioned at the beginning of this section – heap-based sketches find the top-k elements in a stream. In contrast, the previous algorithms only guarantee to find elements whose frequency exceeds a predefined (relative) threshold.

[CCFC04] first came up with the idea to use heap-based sketches to find the top-k elements in a data stream. When they introduced the CountSketch algorithm, the authors additionally proposed to combine it with a heap that keeps track of the k most frequent elements as

depicted in Figure 2.4. Though they only analyzed it for CountSketch, the approach also works with any other sketch-based algorithm. E.g., [RKA16] applies it to CountMin Sketches.

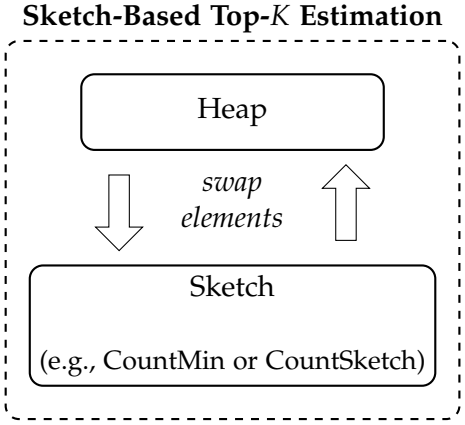


Figure 2.4.: Framework for sketch-based algorithms for top-k predictions that combines a heap of size k with a sketch implementation.

The general framework sets up two data structures: a sketch that estimates frequencies and a heap that tracks the top-k elements. The mechanism works as described in Algorithm 3: Whenever a new element arrives, increment its count if it is in the heap. If the element is not in the heap yet, but it has some space left, add the new element with a count of one. If the heap is full, retrieve the current estimate of the incoming element from the sketch. If its estimate, combined with the current update, exceeds the smallest count in the heap, switch the elements: Write back the outgoing element to the sketch and update the sketch accordingly. Next, add the incoming element in the heap. Set its counter to its estimate from the sketch plus the current update. Otherwise, update the sketch for the incoming element and continue.

To support accurate swaps between the heap and sketch, the algorithm maintains an additional counter for each heap position. This counter tracks how often the element has appeared since it was last inserted into the heap. When the element has to leave the heap, the algorithm uses this value to update the sketch properly. This mechanism guarantees consistency: From the sketch's perspective, whether an element was temporarily in the heap or not makes no difference.

The amount of space the algorithm requires depends on the sketch algorithm used.

Algorithm 3 Framework for Heap-Based Sketch Algorithms

Require: Stream of elements $1, 2, \ldots, n$; heap capacity k
1: Initialize sketch data structure $\mathcal{S}$
2: Initialize empty heap H with capacity k
3: **for** each element i in the stream **do**
4: **if** $i \in H$ **then**
5: $C[i] \leftarrow C[i] + 1$
6: **else if** $|H| < k$ **then**
7: $C[i] \leftarrow 1$
8: **else**
9: Let $\hat{f}_i \leftarrow \mathcal{S}.\text{Estimate}(i)$
10: Let $(y, C_{\text{min}}) \leftarrow$ element y in H with the minimum count C_{min}
11: **if** $\hat{f}_i + 1 > C_{\text{min}}$ **then**
12: $\mathcal{S}.\text{Update}(y, C_{\text{diff}}[y])$
13: $C[i] \leftarrow \hat{f}_i + 1$
14: **else**
15: $\mathcal{S}.\text{Update}(i, 1)$

2.4. Intermediate Overview of Presented Algorithms

The previous sections cover three different types of algorithms as listed in table 2.1. The first class, counter-based algorithms, includes Misra-Gries (c.f. Section 2.3.1) and SpaceSaving (cf. Section 2.3.2), which are deterministic. They were primarily designed for the Heavy Hitters problem. However, they can also be adjusted to estimate frequencies as described in Section 2.2.3.

CountMin (cf. 2.2.1) and CountSketch (cf. 2.2.2), on the other hand, are algorithms that rely on hashing and are, therefore, random. They belong to the class of sketch-based algorithms, also called hash-based algorithms. They were primarily designed for frequency estimation. The CountSketch++ algorithm, which we will discuss in Chapter 3 falls in this class too.

When combining them with an additional heap, it is possible to use them for the top-k problems as well. As introduced in Section 2.3.3, we refer to this class of adjusted algorithms as heap-based algorithms/sketches.

Counter-Based Algorithms	**Sketch-Based Algorithms** *aka Hash-Based Algorithms*	**Heap-Based Algorithms**
• Misra-Gries • SpaceSaving	• CountMin • CountSketch • CountSketch++	• CountMin + Heap • CountSketch + Heap • CountSketch++ + Heap
Primary Use: Heavy Hitters	**Primary Use:** Frequency Estimation	**Primary Use:** Top-k

Table 2.1.: Overview of algorithms for frequency estimation, Heavy Hitters, and top-k estimation discussed in Chapter 2.

3. Enhanced Frequency Estimation With Oracles

In [Aam+23], Aamand et. al. propose CountSketch++, a novel algorithm for frequency estimation based on the classic CountSketch. In addition to the standard setting without predictions, they introduce a learned version of CountSketch++ that uses predictions from an oracle that is able to tell the top-k elements in the data stream. We refer to the setting without predictions simply as CountSketch++ and to the version with predictions as learned CountSketch++. In this chapter, we first describe the algorithm and improve the theoretical analysis of the error bounds of (standard) CountSketch++. Furthermore, we analyze its shortcomings, including the strong assumptions on the oracle for the learned version, and propose alternative approaches to address these issues. As mentioned previously, we focus on Zipfian data streams.

3.1. CountSketch++

As we will describe in more detail in Section 3.1.1, CountSketch++ uses multiple CountSketch tables of two different sizes. Equivalently to the learned version of CountMin and CountSketch, the learned version of CountSketch++ assumes that the algorithm has access to an oracle that can tell whether a given element is in the top-k elements of the data stream or not. Both versions of CountSketch++ outperform standard CountSketch and CountMin and their learned counterparts in the theoretical analysis on Zipfian data.

Algorithm	**Weighted Error**	**Uses Predictions?**
CountMin	$\Theta\left(\frac{N}{B}\right)$	No
CountSketch	$\Theta\left(\frac{N}{\log(n)} \cdot \frac{1}{B}\right)$	No
Learned CountMin	$\Theta\left(\frac{N}{\log^2(n)} \cdot \frac{(\log^2(n/B))}{B}\right)$	Yes
Learned CountSketch	$\Theta\left(\frac{N}{\log^2(n)} \cdot \frac{\log(n/B)}{B}\right)$	Yes
CountSketch++	$\mathcal{O}\left(\frac{N}{\log^2(n)} \cdot \frac{\log B + \text{poly}(\log\log n)}{B}\right)$	No
CountSketch++ *(Improved Bound)*	$\mathcal{O}\left(\frac{N}{\log^2(n)} \cdot \frac{\log B}{B}\right)$	No
Learned CountSketch++	$\mathcal{O}\left(\frac{N}{\log^2(n)} \cdot \frac{1}{B}\right)$	Yes

Table 3.1.: Comparison of weighted error bounds per element for B words of memory and Zipfian frequencies, i.e. $f_i \propto \frac{1}{i}$. The table is taken and adapted from [Aam+23].

3.1.1. Algorithm Description

The algorithm is based on one key observation that holds for any frequency estimation algorithm: If the true frequency of an element is smaller than the expected additive error from the data structure used, outputting zero for these elements yields a lower error than returning the frequency estimate from the data structure. The central question is therefore: How to determine if an element is rare enough so that it is preferable to output zero, while the whole point of the algorithm is that frequencies are unknown? To tackle the problem, [Aam+23] proposes a twofold data structure: The first part is responsible for detecting and filtering out rare elements, while the second part is used to estimate frequencies of the remaining heavier elements.

CountSketch++ For the setting without predictions each of the two parts (filtering and estimating) uses half of the available space budget B. As depicted in Figure 3.1, the first part responsible for filtering rare elements consists of multiple smaller CountSketch tables. More specifically, there are $T = \log\log n$ tables each of size $3 \times \frac{B}{6 \log\log n}$, where n is the number of distinct elements and B denotes the space budget. The second part is one big CountSketch table with three rows and $\frac{B}{6}$ columns. Remember that a constant number of rows is asymptotically optimal for CountSketches with Zipfian data streams [Aam+23]. When an element i arrives with frequency increment Δ, the algorithm inserts the element in each of the small CountSketch tables as well as in the big CountSketch table according to the standard CountSketch procedure (cf. algorithm 4). The query operation comprises two steps: First, the algorithm queries all of the small CountSketch tables according to standard CountSketch rules and takes the median over these estimates (cf. Algorithm 5, l. 2-4). We refer to this median over medians as $\tilde{f}_i$ for element i. If this median is smaller than a predefined threshold τ, the algorithm assumes that the element is rare and returns zero (cf. Algorithm 5, l. 5f). Otherwise, it queries the big CountSketch table and returns the resulting estimate (cf. algorithm 5, l. 8). [Aam+23] shows that in expectation this mechanism incurs a weighted error upper bounded by $\mathcal{O}\left(\frac{N}{\log(n)} \cdot \frac{\log B + \text{poly}(\log\log n)}{B \log n}\right)$ for $B \geq \log n$.

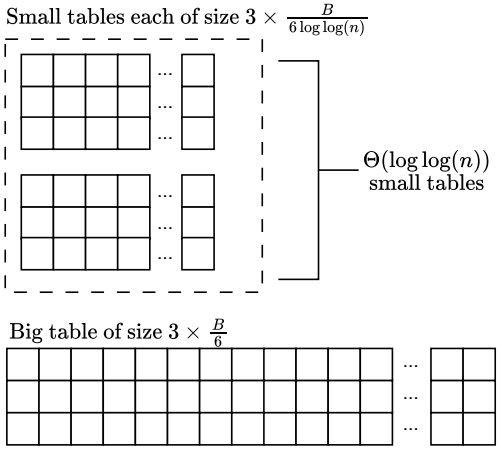


Figure 3.1.: The CountSketch++ data structure consists of multiple small and one big table.

Algorithm 4 CountSketch++ Update Algorithm *Source: [Aam+23]*

Require: Stream of updates to an n-dimensional vector, space budget B

1: **procedure** UPDATE
2: $T \leftarrow \Theta(\log \log n)$
3: **for** $j = 1$ to $T - 1$ **do**
4: $S_j \leftarrow$ CountSketch table with 3 rows and $\frac{B}{6T}$ columns
5: $S_T \leftarrow$ CountSketch table with 3 rows and $\frac{B}{6}$ columns
6: **for** stream element (i, Δ) **do**
7: Input (i, Δ) in each of the T CountSketch tables S_j

Algorithm 5 CountSketch++ Query Algorithm *Source: [Aam+23]*

Require: Index $i \in [n]$ for which we want to estimate f_i

1: **procedure** QUERY
2: **for** $j = 0$ to $T - 1$ **do**
3: $\hat{f}_i^j \leftarrow$ estimate of the i^{th} frequency given by table S_j
4: $\tilde{f}_i \leftarrow \text{Median}(\hat{f}_i^0, \ldots, \hat{f}_i^{T-1})$
5: **if** $\tilde{f}_i < \tau$ **then**
6: **return** 0
7: **else**
8: **return** $\hat{f}_i^T$, the estimate given by table S_T

Learned CountSketch++ In addition to the two parts in the version without predictions for filtering rare elements and estimating the remaining frequencies, the learned CountSketch++ maintains a separate array for the top-k elements as shown in Figure 3.2. Given space budget B, the algorithm uses half of the space to track exact counts for the top-$\frac{B}{2}$ elements. The other half remains to set up the data structures as described for the standard version without predictions. The $\log B$ small tables are now of size $3x\frac{B}{12 \log \log n}$. The big table has three rows and $\frac{B}{12}$ columns. If one of the top-$\frac{B}{2}$ elements arrives, the algorithm updates its (exact) counter in the separate array (cf. Algorithm 6, l.8-9). The rest of the data structure remains unchanged. For all other elements, the algorithm behaves as described for the version without predictions. During the query operation, the algorithm first checks whether the element is in the heap tracking the most frequent elements. If so, it returns the exact count stored there (cf. Algorithm 7, l. 2-4). Otherwise, the algorithm again behaves as standard CountSketch++. The upper bound of the weighted error improves to $\mathcal{O}\left(\frac{N}{\log(n)} \cdot \frac{1}{B \log n}\right)$ compared to the standard version without predictions.

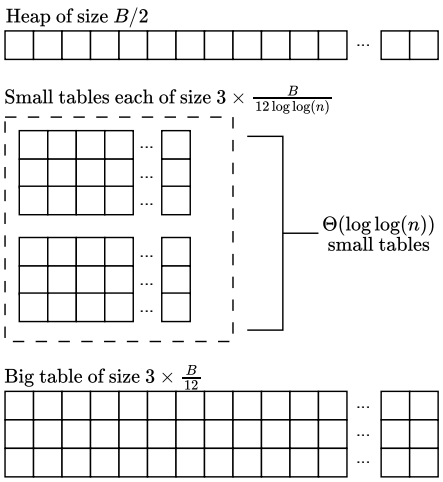


Figure 3.2.: The learned CountSketch++ data structure consists of multiple small, one big table, and a heap to track the predicted top-k elements.

Algorithm 6 Learned CountSketch++ Update Algorithm *Source: [Aam+23]*

Input: Stream of updates to an n dimensional vector, space budget B, access to a Heavy Hitter oracle which correctly identifies the top-$B/2$ Heavy Hitters
procedure UPDATE
    $T \leftarrow O(\log \log n)$
    for $j = 0$ to $T - 1$ **do**
        $S_j \leftarrow$ CountSketch table with 3 rows and $\frac{B}{12T}$ columns
    $S_T \leftarrow$ CountSketch table with 3 rows and $\frac{B}{12}$ columns
    for stream element (i, Δ) **do**
        if i is a top-$B/2$ Heavy Hitter **then**
            Maintain the frequency of i exactly
        else
            Input (i, Δ) in each of the T CountSketch tables S_j

Algorithm 7 Learned CountSketch++ Query Algorithm *Source: [Aam+23]*

Input: Index $i \in [n]$ for which we want to estimate f_i
procedure QUERY
    if i is a top-$B/2$ Heavy Hitter **then**
        Output the exact maintained frequency of i
    else
        return $\hat{f}_i \leftarrow$ output of Alg. 5 using the CountSketch tables created in Alg. 6

3.1.2. Analysis

Intuitively, the CountSketch++ algorithm – or rather its mathematical analysis – partitions the elements into three groups as shown in Figure 3.3. The first group consists of the most frequent elements. Their high frequencies ensure, that with very high probability, the algorithm correctly labels them as Heavy Hitters during the filtering step. The second group comprises the very rare elements. Like for elements in group one, the algorithm's filter mechanism most likely succeeds to label them correctly. For group two elements, this means the algorithm labels them as rare elements in the filtering step and returns zero w.h.p. The third group includes all remaining elements that are neither particularly frequent nor especially rare. Their frequencies are close to the threshold τ used to determine whether the algorithm outputs zero or looks up the frequency estimate in the big table. As the improved theoretical analysis in Section 3.2 shows, the highest error for these elements happens if the algorithm classifies them as rare elements and returns zero.

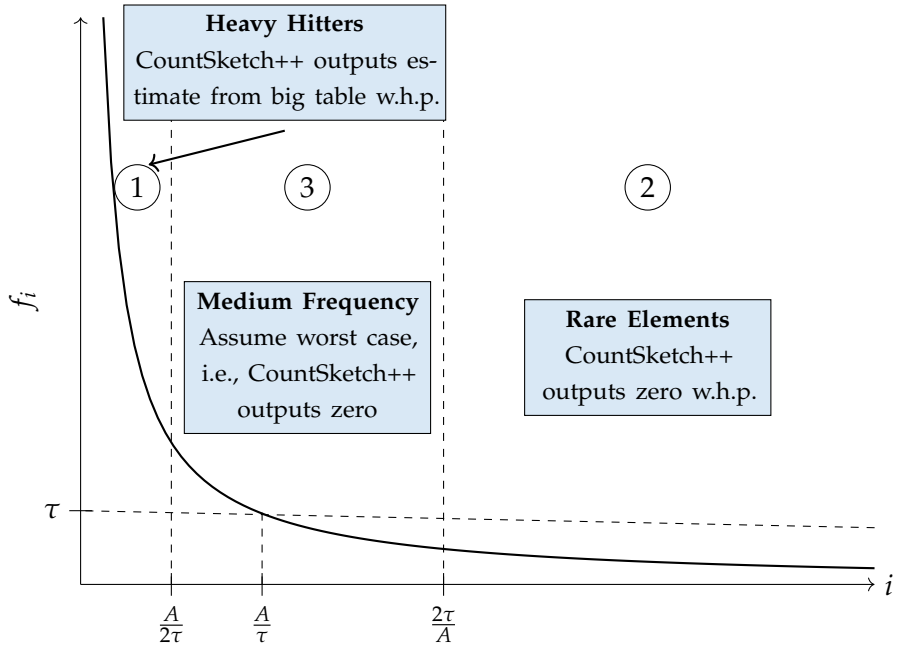


Figure 3.3.: Intuitively, the CountSketch++ algorithm splits the elements into three groups depending on their frequency relative to the threshold τ used to filter rare elements.

To understand which components of the algorithm cause the error to decrease compared to a CountSketch, we begin by examining the CountSketch++ without predictions. Here, the improvement arises solely from returning zero for rare elements. In fact, the error for all other elements remains unchanged on the order of $\mathcal{O}\left(\frac{A}{B}\right)$. This happens because CountSketch++ still uses the estimate directly from the large CountSketch table whenever an element is not labeled as rare by the filtering step. In other words, CountSketch++'s additional filtering step at query time does not change the error's order of magnitude for elements labeled as frequent by the algorithm. Estimates from the big table remain those of a standard CountSketch table. On the other hand, the additional improvement in the learned compared to the standard version of CountSketch++ solely comes from having exact estimates for the top-k. For all other elements, the error remains the same as for the standard version without predictions.

This insight entails two further observations:

First, even though the algorithm stores exact counts for the top-$\frac{B}{2}$ elements, and does not insert these highest frequencies into the sketch, the error for the remaining elements does not improve asymptotically. The intuition behind this is that there are still significantly more than B elements in the order of magnitude of $\mathcal{O}\left(\frac{A}{B}\right)$. In expectation, one would therefore assume more than one such element per hash bucket with $f_i = \Theta\left(\frac{A}{B}\right)$. Consequently, no matter which element in a hash bucket we consider, we would always expect enough elements in the same bucket of the order of magnitude of $\mathcal{O}\left(\frac{A}{B}\right)$, keeping the error at $\mathcal{O}\left(\frac{A}{B}\right)$.

The second observation results from the proof in [Aam+23] for the upper bound of the weighted error for the learned CountSketch++: Taking away all top-$\frac{B}{2}$ elements and tracking their counts exactly, only leaves elements from group three to be stored in the sketch. I.e., the only elements remaining are those whose frequency is so small that w.h.p. the algorithm labels them as rare during the filtering step and returns zero. This observation suggests a natural simplification: One could simply omit the sketch entirely. Instead, only track exact counts for top-$\frac{B}{2}$ elements, and return a frequency estimate of zero for all other elements. We come back to this observation later in Section 3.3.1 and prove that indeed, this mechanism asymptotically achieves the same error. The analysis confirms that assuming a perfect oracle that knows that top-$\frac{B}{2}$ elements makes this trivial strategy already achieve state-of-the-art performance.

The tight bounds for the CountSketch algorithm for Zipfian data proven in [Aam+23] show that the order of magnitude of the error is inversely proportional to the width of the sketch tables. Since the same work shows that a constant number of rows is asymptotically optimal, one can design a CountSketch table where the error is asymptotically $\mathcal{O}\left(\frac{A}{B}\right)$, while maintaining a space complexity of $\mathcal{O}(B)$. To accurately distinguish between scarcer and heavier elements in a sketch, the error intervals around their true frequencies may not overlap. Therefore, all elements whose frequency is smaller than or asymptotically equal to the sketch's expected error are, in effect, indistinguishable from each other in their estimated frequencies. For such elements, the CountSketch incurs, in expectation, an error of $\mathcal{O}\left(\frac{A}{B}\right)$, leading to an estimated frequency of $\hat{f}_i = \Theta\left(\frac{A}{B}\right)$.

Figure 3.4 illustrates this point for elements a and b with frequencies $f_a = \frac{A}{B \log B}$ and $f_b = \frac{A}{B}$. The shaded regions around the true frequencies indicate the intervals within which the algorithm's estimates fall with high probability. As these intervals largely overlap, the algorithm can not reliably tell in which order of magnitudes these elements are, or which of the elements a and b is heavier, although their true frequencies differ asymptotically: $\mathcal{O}\left(\frac{A}{B \log B}\right)$ versus $\mathcal{O}\left(\frac{A}{B}\right)$ respectively.

In practical terms, this observation implies that the set of elements a CountSketch++ can reliably filter is inherently tied to the width of its smaller sketch tables. We return to this observation in Section 3.2.1, where we demonstrate that increasing the memory budget of the standard CountSketch++ by a logarithmic factor compared to the learned version leads to the same asymptotic error, even without using predictions.

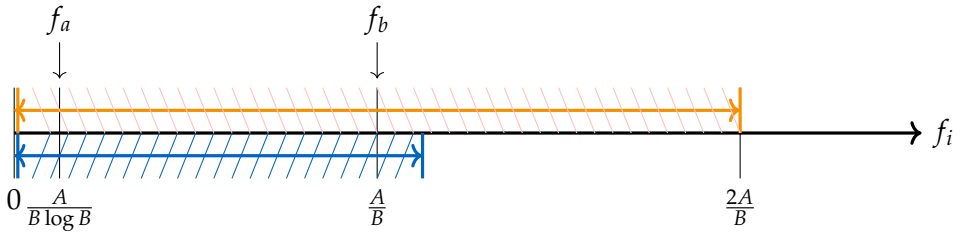


Figure 3.4.: If the error intervals around the true frequencies overlap (orange and blue shaded areas), the CountSketch++ algorithm can not distinguish the estimates order or frequency reliably.

3.2. Improved Analysis of CountSketch++ Without Predictions

In this section, we improve the analysis for the upper bound that, achieves a better result for the lower weighted error for the CountSketch++ algorithm without predictions. The intuition follows the proof technique in [Aam+23]. More specifically, we prove the following theorem:

Theorem 3.2.1

Consider the CountSketch++ algorithm as described in algorithms 4 and 7 with space parameter $B \geq \log n$ updated over a Zipfian stream. The weighted error is:

$$\frac{1}{N} \sum_{i \in [n]} f_i \cdot |\hat{f}_i - f_i| = \mathcal{O}\left(\frac{N}{\log^2(n)} \cdot \frac{\log B}{B}\right)$$

Proof Intuition Remember the intuition to partition the elements into three groups (cf. Figure 3.3): Group one contains the heaviest elements. Group two comprises the rarest elements. Finally, group three holds all elements in between whose frequencies are close to the threshold τ used to determine whether an element is heavy or not during the filtering step. Reducing the number of elements in group three, we can reduce the expression from $\mathcal{O}(\frac{N}{\log^2(n)} \cdot \frac{\log B + poly(\log \log n)}{B})$ to $\mathcal{O}(\frac{N}{\log^2(n)} \cdot \frac{\log B}{B})$. To shrink group three, we keep the threshold τ as suggested in [Aam+23] but tighten the boundaries from originally $\left(\frac{A}{\tau}, \frac{B}{(\log \log n)^4}\right)$ to $\left(\frac{A}{2\tau}, \frac{2A}{\tau} = \frac{2B}{\mathcal{O}(\log \log n)}\right)$. This improves the original error bound by a factor of $\frac{1}{poly(\log \log n)}$, where n is the number of distinct elements in the stream.

Given a threshold of $\tau = \frac{A \cdot \mathcal{O}(\log \log n)}{B}$, we show that for any element $i : f_i \leq \frac{\tau}{2}$, the algorithm outputs zero with very high probability. For any element $i : f_i \geq 2\tau$, the algorithm outputs the estimate in the big sketch table w.h.p. For any element in between these two thresholds, we assume the worst case, that is, that the algorithm outputs zero. Remember that outputting zero incurs an error of $\frac{A}{i}$, whereas outputting the estimate from the big sketch table results in an error of $\frac{A}{B}$ (cf. [Aam+23], Theorem C.4).

Proof We are going to use the following theorem as stated and proofed in [Aam+23]:

Theorem 3.2.2 *Source: [Aam+23]*

Let $\hat{f}_i$ be the estimate of the i^{th} frequency given by a 3 x $\frac{B'}{3}$ CountSketch table, where $B' \geq \log(n)$. There exists a universal constant C such that the following inequality holds:

$$\forall t \geq 3/B', \qquad Pr[|f_i - \hat{f}_i| \geq t] \leq C' \left(\frac{\log(tB')}{tB'}\right)^2$$

First, we decompose the total expected weighted error, $\mathbf{E_{total}}$ in three parts. Each part represents the error incurred by the respective group of elements, where group one contains Heavy Hitters, group two represents rare elements, and group three the elements in between (cf. Figure 3.3):

$$E_{total} = \frac{1}{N} \sum_{i \in [n]} f_i \cdot |\hat{f}_i - f_i| = \frac{1}{N}(E_1 + E_2 + E_3)$$

For all groups of elements, let us decompose the expected weighted error $\mathbf{E_x}, \mathbf{x} \in \{\mathbf{1}, \mathbf{2}, \mathbf{3}\}$ they contribute to the final error E_{total} into two parts E_S and E_F. $\mathbf{E_S}$ describes the error obtained if we output the estimate in the sketch. $\mathbf{E_F}$ is the error incurred by outputting zero. Weighting each part, E_S and E_F, by the probability that these events occur (i.e., outputting zero vs. outputting the estimate from the big sketch table) leads to:

$$E_x = \sum_{i \in [G_x]} (p_S(G_x) \cdot E_S + p_F(G_x) \cdot E_F)$$

where $\mathbf{G_x}, \mathbf{x} \in \{\mathbf{1}, \mathbf{2}, \mathbf{3}\}$ is the group of elements under consideration, $\mathbf{p_S}(\mathbf{G_x})$ is the probability that the algorithm outputs the estimate from the big table for elements in group G_x, and $\mathbf{p_F}(\mathbf{G_x})$ is the probability that the algorithm outputs zero. Additionally, we know that for all groups:

$$E_S = \mathbb{E}[f_i \cdot |f_i - \tilde{f}_i|] = \frac{A}{i} \cdot \frac{A}{B} = \frac{A^2}{iB}$$

$$E_F = \mathbb{E}[f_i \cdot |f_i - 0|] = \frac{1}{i} \cdot \frac{A}{i} = \frac{A}{i^2}$$

Since for all $i \leq B$ it holds that $E_S = \frac{A}{i} \cdot \frac{A}{B} \leq \frac{A^2}{i^2} = E_F$, it is desirable to output the estimate from the big CountSketch table $\tilde{f}_i$ for these elements. For all other elements it follows that it is preferable to output zero.

Case 1: This case deals with all elements i, such that

$$i \leq \frac{A}{2\tau} = \frac{B}{2 \log\log n} \rightarrow f_i = \frac{A}{i} \geq 2\tau = \frac{2A \log\log n}{B}$$

For these elements, it is desirable that the algorithm outputs the estimate from the big sketch table. Accordingly, the bad scenario is that the algorithm labels these frequent elements as rare and outputs zero, since

$$\forall i \leq \frac{A}{2\tau}: \qquad f_i = \frac{A}{i} \geq \frac{2A \log\log n}{B} > \frac{A}{B}$$

, where $\frac{A}{B}$ is the expected additive error incurred by a CountSketch table with B columns. To

upper bound the error E_1, we aim to minimize the upper bound of the bad scenario, i.e., the probability that the algorithm outputs zero, p_F. Let $\hat{f}_i^{(k)}$ be the frequency estimate of the i^{th} element given by the k^{th} small CountSketch table. In the following, $\lesssim$ denotes inequality up to a constant factor. The probability that a single estimate from a small table $\hat{f}_i^{(k)}$ is smaller than τ is:

$$
\begin{aligned}
\Pr[\hat{f}_i^{(k)} \leq \tau] &= \Pr\left[-\hat{f}_i^{(k)} \geq -\tau\right] && \textit{Multiply both sides by -1} \\
&= \Pr\left[f_i - \hat{f}_i^{(k)} \geq f_i - \tau\right] && \textit{Add } f_i \textit{ to both sides} \\
&\leq \Pr\left[|f_i - \hat{f}_i^{(k)}| \geq f_i - \tau\right] && \textit{Take the absolute value of the left side} \\
&\leq \Pr\left[|f_i - \hat{f}_i^{(k)}| \geq 2\tau - \tau\right] && \textit{Lower bound } f_i \textit{ by } 2\tau\textit{, the minimum frequency in group 1} \\
&\leq \Pr\left[|f_i - \hat{f}_i^{(k)}| \geq \tau\right]
\end{aligned}
$$

Given that term, we can now apply Theorem 3.2.2:

$$
\begin{aligned}
\Pr\left[|f_i - \hat{f}_i^{(k)}| \geq \tau\right] &\leq C' \left(\frac{\log(\tau \mathcal{B}')}{\tau \mathcal{B}'}\right)^2 && \textit{Apply Theorem 3.2.2} \\
&\leq C' \left(\frac{\log(\frac{CA \cdot \log\log n}{B} \cdot \frac{B}{2T})}{\frac{CA \cdot \log\log n}{B} \cdot \frac{B}{2T}}\right)^2 && \textit{Plug in values: } \tau = \frac{A \cdot \mathcal{O}(\log\log n)}{B} \textit{ and } B' = \frac{B}{2T} \\
&\leq C' \left(\frac{\log(\frac{CA \cdot \log\log n}{B} \cdot \frac{B}{2C'' \log\log n})}{\frac{CA \cdot \log\log n}{B} \cdot \frac{B}{2C'' \log\log n}}\right)^2 && \textit{Plug in value for } T, T = \mathcal{O}(\log\log n) \\
&\leq C' \left(\frac{2C'' \log\left(\frac{AC}{2C''}\right)}{AC}\right)^2 && \textit{Simplify terms where possible} \\
&\leq 4C'C''^2 \left(\frac{\log\left(\frac{AC}{2C''}\right)}{AC}\right)^2 && \textit{Factor out } 2C'' \\
&\lesssim \left(\frac{\log(AC)}{AC}\right)^2 \lesssim \frac{1}{8} && \textit{Omit constant parts and upper bound by } \frac{1}{8}
\end{aligned}
$$

We have now shown, that the probability that a single small CountSketch table in the algorithm labels a heavy element as rare is at most $\frac{1}{8}$. Next, we want to upper bound the probability that the filtering step as a whole labels a heavy element as rare. This happens if at least half of the T small tables return an estimate smaller than τ. Let m be the number of small tables with estimates smaller than τ. For the CountSketch++ to label an element as rare it must hold that

$m \geq \frac{T}{2}$. There are $\binom{T}{m}$ different combinations to choose m from the T estimates. Given that the estimates are independent, and the probability that a single estimate from a small table is smaller than τ is upper bounded by $\frac{1}{8}$ as stated above, we get:

$$
\begin{aligned}
p_F(G_1) &= \sum_{m=T/2}^{T} \binom{T}{m} \cdot Pr[\hat{f}_i^{(k)} \leq \tau]^m \cdot Pr[\hat{f}_i^{(k)} > \tau]^{T-m} \\
&\leq \sum_{m=T/2}^{T} \binom{T}{m} \cdot Pr[\hat{f}_i^{(k)} \leq \tau]^m && \textit{Omit } Pr[\hat{f}_i^{(k)} > \tau]^{T-m} \\
&\leq \frac{T}{2} \cdot \binom{T}{T/2} \cdot Pr[\hat{f}_i^{(k)} \leq \tau]^{T/2} && \textit{Upper bound the single addends } \left(\textit{max. for } m = \frac{T}{2}\right) \\
&\leq \frac{T}{2} \cdot \left(\frac{2eT}{T}\right)^{T/2} \cdot Pr[\hat{f}_i^{(k)} \leq \tau]^{T/2} && \textit{Upper bound binom. coefficient by } \binom{n}{k} \leq \left(\frac{en}{k}\right)^k \\
&\leq \frac{T}{2} \cdot (2e)^{T/2} \cdot Pr[\hat{f}_i^{(k)} \leq \tau]^{T/2} && \textit{Simplify terms} \\
&\lesssim \frac{T}{2} \cdot (2e)^{T/2} \cdot \left(\frac{1}{8}\right)^{\frac{T}{2}} && \textit{Plug in upper bound for } Pr[\hat{f}_i^{(k)} \leq \tau] \\
&\lesssim \frac{T}{2} \cdot (2e)^{T/2} \cdot e^{-\ln(8)\frac{T}{2}} && \textit{Use trick that } \frac{1}{8} = e^{-\ln(8)} \\
&\lesssim \frac{T}{2} \cdot (2e)^{T/2} \cdot e^{-\Theta(T)} && \textit{Rewrite } e^{-\ln(8)\frac{T}{2}} \textit{ as } e^{-\Theta(T)} \\
&\lesssim e^{-\Theta(T)} && \textit{Omit further constant parts of term} \\
&\lesssim e^{-\Theta(\log\log n)} && \textit{Plug in value for } T \\
&\lesssim \frac{1}{\log^C(n)} && \textit{Rewrite term; plug in constant } C \textit{ for } \Theta \\
&\lesssim \frac{1}{\log^{100}(n)} && \textit{Plug in concrete value for } C
\end{aligned}
$$

Plugging in the obtained upper bound for $p_F(G_1)$, the probability that the bad scenario occurs, i.e., the algorithm outputs zero, and upper-bounding the probability for the good scenario

$p_S(G_1)$ by one, we get:

$$
\begin{aligned}
E_1 &= \sum_{i=1}^{\frac{A}{2\tau}} \mathbb{E}[f_i \cdot |f_i - \hat{f}_i|] \\
&= \sum_{i=1}^{\frac{A}{2\tau}} \left(p_F(G_1) \cdot E_F + p_S(G_1) \cdot E_S\right) \\
&\lesssim \sum_{i=1}^{\frac{A}{2\tau}} \left(p_F(G_1) \cdot \frac{A^2}{i^2} + p_S(G_1) \cdot \frac{A^2}{iB}\right) && \text{Plug in } E_S = \frac{A^2}{iB} \text{ and } E_F = \frac{A^2}{i^2} \\
&\lesssim \sum_{i=1}^{\frac{A}{2\tau}} \left(\frac{1}{\log^{100}(n)} \cdot \frac{A^2}{i^2} + \frac{A^2}{iB}\right) && \text{Plug in obtained upper bound for } p_F(G_1) \\
& && \text{and upper bound } p_S(G_1) \text{ by one} \\
&\lesssim \frac{A^2}{\log^{100}(n)} \sum_{i=1}^{\frac{A}{2\tau}} \frac{1}{i^2} + \frac{A^2}{B} \sum_{i=1}^{\frac{A}{2\tau}} \frac{1}{i} && \text{Factor out constants} \\
&\lesssim \frac{A^2}{\log^{100}(n)} \int_1^{\frac{A}{2\tau}} \frac{1}{x^2}\, dx + \frac{A^2}{B} \cdot \log\left(\frac{A}{2\tau}\right) && \text{Upper bound left sum by integral and right sum} \\
& && \text{by harmonic series } H(n) = \sum_{x=1}^{n} \frac{1}{x} \lesssim \log(n) \\
&\lesssim \frac{A^2}{\log^{100}(n)} \cdot \left(1 - \frac{2\tau}{A}\right) + \frac{A^2}{B} \cdot \log\left(\frac{A}{2\tau}\right) && \text{Calculate integral} \\
&\lesssim \frac{A^2}{\log^{100}(n)} \cdot \left(1 - \frac{2\log\log n}{B}\right) + \frac{A^2}{B} \cdot \log\left(\frac{B}{2\log\log n}\right) && \text{Plug in value for } \tau \\
&\lesssim A^2 \left(\frac{B - 2\log\log n}{\log^{100}(n) B} + \frac{\log B}{B} - \frac{\log(2\log\log n)}{B}\right) && \text{Factor out} A^2 \text{ and pull apart logarithmic term} \\
&= \mathcal{O}\left(\frac{A^2 \log B}{B}\right) && \text{Only keep dominating part}
\end{aligned}
$$

Case 2: This case deals with all elements i, such that

$$i \geq \frac{2A}{\tau} = \frac{2B}{\log\log n} \rightarrow f_i = \frac{A}{i} \leq \frac{\tau}{2} = \frac{A \log\log n}{2B}$$

Note that not for all elements in this group, the best and worst cases are the same: For some elements i such that $\frac{2B}{\log\log n} \leq i \leq B$, it is indeed still preferable to output the estimate from the big sketch table. However, for most elements in group two, the best and worst case scenarios are just inverse to the first case: It is desirable that the algorithm outputs

zero, while the bad scenario is that the algorithm outputs the estimate from the big sketch table.

Following the same reasoning as in the first case, we can upper bound the error E_2 by upper bounding the bad event, i.e., the probability that the algorithm outputs the estimate from the big sketch table, p_S. To do so, let us first consider the probability that a single estimate from the small tables is larger than τ:

$$\begin{aligned}
\Pr[\hat{f}_i^{(k)} \geq \tau] &= \Pr\left[\hat{f}_i^{(k)} - f_i \geq \tau - f_i\right] && \textit{Subtract } f_i \textit{ from both sides} \\
&\leq \Pr\left[|\hat{f}_i^{(k)} - f_i| \geq \tau - f_i\right] && \textit{Take the absolute value of the left side} \\
&\leq \Pr\left[|\hat{f}_i^{(k)} - f_i| \geq \tau - \frac{\tau}{2}\right] && \textit{Upper bound } f_i \textit{ by } \frac{\tau}{2}\textit{, the max. frequency in group 2} \\
&\leq \Pr\left[|\hat{f}_i^{(k)} - f_i| \geq \frac{\tau}{2}\right]
\end{aligned}$$

Like in the first case, we can now apply Theorem 3.2.2:

$$\begin{aligned}
\Pr\left[|f_i - \hat{f}_i^{(k)}| \geq \frac{\tau}{2}\right] &\leq C'\left(\frac{\log(\frac{\tau}{2}B')}{\frac{\tau}{2}B'}\right)^2 && \textit{Apply Theorem 3.2.2} \\
&\leq C'\left(\frac{\log(\frac{CA\log\log n}{2B}\cdot\frac{B}{2T})}{\frac{CA\cdot\log\log n}{2B}\cdot\frac{B}{2T}}\right)^2 && \textit{Plug in values: } \tau = \frac{A\mathcal{O}(\log\log n)}{B} \textit{ and } B' = \frac{B}{2T} \\
&\leq C'\left(\frac{\log(\frac{CA\log\log n}{2B}\cdot\frac{B}{2C''\log\log n})}{\frac{CA\cdot\log\log n}{2B}\cdot\frac{B}{2C''\log\log n}}\right)^2 && \textit{Plug in value for } T, T = \mathcal{O}(\log\log n) \\
&\leq C'\left(\frac{4C''\log\left(\frac{AC}{4C''}\right)}{AC}\right)^2 && \textit{Simplify terms where possible} \\
&\leq 16C'C''^2\left(\frac{\log\left(\frac{AC}{4C''}\right)}{AC}\right)^2 && \textit{Factor out constants} \\
&\lesssim \left(\frac{\log(AC)}{AC}\right)^2 \lesssim \frac{1}{8} && \textit{Omit irrelevant constants and upper bound by } \frac{1}{8}
\end{aligned}$$

Now that we have upper bounded the probability that a single small table fails and labels a rare element as heavy, we can determine the probability that the filtering step as a whole labels a heavy element as rare. That happens, if at least half of the T small tables return an estimate larger than τ. Let m be the number of small tables with estimates larger than τ. The

steps follow the same logic as in case one:

$$
\begin{aligned}
p_S(G_2) &= \sum_{m=T/2}^{T} \binom{T}{m} \cdot Pr[\hat{f}_i^{(k)} \geq \tau]^m \cdot Pr[\hat{f}_i^{(k)} < \tau]^{T-m} \\
&\leq \sum_{m=T/2}^{T} \binom{T}{m} \cdot Pr[\hat{f}_i^{(k)} \geq \tau]^m && \textit{Omit } Pr[\hat{f}_i^{(k)} < \tau]^{T-m} \\
&\leq \frac{T}{2} \cdot \binom{T}{T/2} \cdot Pr[\hat{f}_i^{(k)} \geq \tau]^{T/2} && \textit{Upper bound the single addends } \left(\textit{max. for } m = \frac{T}{2}\right) \\
&\leq \frac{T}{2} \cdot \left(\frac{2eT}{T}\right)^{T/2} \cdot Pr[\hat{f}_i^{(k)} \geq \tau]^{T/2} && \textit{Upper bound binom. coefficient by } \binom{n}{k} \leq \left(\frac{en}{k}\right)^k \\
&\leq \frac{T}{2} \cdot (2e)^{T/2} \cdot Pr[\hat{f}_i^{(k)} \geq \tau]^{T/2} && \textit{Simplify terms} \\
&\lesssim \frac{T}{2} \cdot (2e)^{T/2} \cdot \left(\frac{1}{8}\right)^{\frac{T}{2}} && \textit{Plug in upper bound for } Pr[\hat{f}_i^{(k)} \geq \tau] \\
&\lesssim \frac{T}{2} \cdot (2e)^{T/2} \cdot e^{-\ln(8)\frac{T}{2}} && \textit{Use trick that } e^{-\ln(8)} = \frac{1}{8} \\
&\lesssim \frac{T}{2} \cdot (2e)^{T/2} \cdot e^{-\Theta(T)} && \textit{Rewrite } e^{-\ln(8)\frac{T}{2}} \textit{ as } e^{-\Theta(T)} \\
&\lesssim e^{-\Theta(T)} && \textit{Omit further constant parts of term} \\
&\lesssim e^{-\Theta(\log\log n)} && \textit{Plug in value for } T \\
&\lesssim \frac{1}{\log^C(n)} && \textit{Rewrite term; plug in constant } C \textit{ for } \Theta \\
&\lesssim \frac{1}{\log^{100}(n)} && \textit{Plug in concrete value for } C
\end{aligned}
$$

Now that we have found an upper bound for $p_S(G_2)$, the probability that the bad scenario occurs, i.e., the algorithm outputs the estimate from the big sketch table for a rare element, we can upper bound the error E_2. By upper-bounding the probability for the good scenario

$p_F(G_2)$ by one, we get:

$$
\begin{aligned}
E_2 &= \sum_{i=\frac{2A}{\tau}}^{n} \mathbb{E}[f_i \cdot |f_i - \hat{f}_i|] \\
&\lesssim \sum_{i=\frac{2A}{\tau}}^{n} (p_S(G_2) \cdot E_S + p_F(G_2) \cdot E_F) \\
&\lesssim \sum_{i=\frac{2A}{\tau}}^{n} \left(p_S(G_2) \cdot \frac{A^2}{iB} + p_F(G_2) \cdot \frac{A^2}{i^2} \right) && \textit{Plug in } E_S = \frac{A^2}{iB} \textit{ and } E_F = \frac{A^2}{i^2} \\
&\lesssim \sum_{i=\frac{2A}{\tau}}^{n} \left(\frac{A^2}{\log(n)^{100}} \cdot \frac{1}{iB} + \frac{A^2}{i^2} \right) && \textit{Plug in upper bound for } p_S(G_2) \textit{ and upper bound } p_F(G_2) \textit{ by one} \\
&\lesssim \frac{A^2}{B \cdot \log^{100}(n)} \cdot \sum_{i=\frac{2A}{\tau}}^{n} \frac{1}{i} + A^2 \cdot \sum_{i=\frac{2A}{\tau}}^{n} \frac{1}{i^2} && \textit{Split sum and factor out constant parts} \\
&\lesssim \frac{A^2}{B \cdot \log^{100}(n)} \cdot \log(n) + A^2 \cdot \int_{\frac{2A}{\tau}}^{n} \frac{1}{x^2}\, dx && \textit{Upper bound right sum by integral and left side by} \\
& && \textit{harmonic series } H(n) = \sum_{x=1}^{n} \frac{1}{x} \lesssim \log(n) \\
&\lesssim A^2 \cdot \left[\frac{1}{B \cdot \log^{99}(n)} + \left(-\frac{1}{n} + \frac{\tau}{2A} \right) \right] && \textit{Factor out} A^2 \textit{ and calculate integral} \\
&\lesssim A^2 \cdot \left[\frac{1}{B \cdot \log^{99}(n)} + \frac{\log\log n}{B} \right] && \textit{Plug in value for } \tau\textit{, simplify, and omit constant parts} \\
&= \mathcal{O}\left(\frac{A^2 \log\log n}{B} \right) && \textit{Only keep dominating part}
\end{aligned}
$$

Case 3: The last case to consider is the case of all elements i such that

$$\frac{A}{2\tau} \leq i \leq \frac{2A}{\tau} \rightarrow 2\tau \geq f_i \geq \frac{\tau}{2}$$

In the worst case, the algorithm outputs zero for all these elements, since

$$\frac{A}{i} \geq \frac{\tau}{2} \gtrsim \frac{A \log\log n}{2B} > \frac{A}{B}$$

Therefore, we can upper bound the error by assuming the worst-case scenario for all elements

in group three:

$$
\begin{aligned}
E_3 &= \mathbb{E}[f_i \cdot |f_i - \hat{f}_i|] \\
&\lesssim \sum_{i=\frac{A}{2\tau}}^{\frac{2A}{\tau}} (p_S(G_3) \cdot E_S + p_F(G_3) \cdot E_F) \\
&\lesssim \sum_{i=\frac{A}{2\tau}}^{\frac{2A}{\tau}} \frac{A^2}{i^2} \qquad \textit{Assume worst case, i.e. } p_F(G_3) = 1 \\
&\lesssim A^2 \cdot \int_{\frac{A}{2\tau}}^{\frac{2A}{\tau}} \frac{1}{x^2}\, dx \qquad \textit{Upper bound sum by integral} \\
&\lesssim A^2 \left(-\frac{\tau}{2A} + \frac{2\tau}{A} \right) \qquad \textit{Calculate the integral} \\
&\lesssim A^2 \left(-\frac{\log\log n}{2B} + \frac{2\log\log n}{B} \right) \qquad \textit{Plug in value for } \tau \\
&= \mathcal{O}\left(\frac{A^2 \log\log n}{B} \right)
\end{aligned}
$$

Finally, we can combine the three cases to obtain the total expected weighted error:

$$
\begin{aligned}
E_{total} &= \frac{1}{N}(E_1 + E_2 + E_3) = \frac{1}{N}\left[\mathcal{O}\left(\frac{A^2 \log B}{B} \right) + \mathcal{O}\left(\frac{A^2 \log\log n}{B} \right) + \mathcal{O}\left(\frac{A^2 \log\log n}{B} \right) \right] \\
&= \frac{1}{N} \cdot \mathcal{O}\left(\frac{A^2 \log B}{B} \right) = \frac{1}{N} \cdot \mathcal{O}\left(\frac{N^2}{\log^2(n)} \frac{\log B}{B} \right) \\
&= \mathcal{O}\left(\frac{N}{\log^2(n)} \frac{\log B}{B} \right)
\end{aligned}
$$

since $\log B > \log\log n$.

□

This concludes the improved analysis of the weighted error for standard CountSketch++. It establishes an improved upper bound of $\mathcal{O}\left(\frac{N}{\log^2(n)} \frac{\log B}{B}\right)$ as stated in Theorem 3.2.1.

3.2.1. Equalizing Learned CountSketch++ Without Using Predictions

Though the analysis above improves the upper bound by a logarithmic factor, it is still asymptotically larger than the upper bound for the weighted error of the learned CountSketch++ algorithm, which is $\mathcal{O}\left(\frac{N}{\log^2(n)} \cdot \frac{1}{B}\right)$. As we will argue later in Section 3.3.2 in more detail, a perfect oracle might not be available for all real-world settings. An easy way to work

around this problem is to increase the space budget of the standard CountSketch++ algorithm without predictions to achieve the same precision as the learned version. The question is therefore: How much do we need to increase the space budget of a standard CountSketch++ to achieve this? Let B denote the space budget of the learned CountSketch++, which achieves a weighted error of $\frac{N}{\log^2(n)} \cdot \frac{1}{B}$. We want to find a space budget B' for the CountSketch++ without predictions such that the weighted error is also $\frac{N}{\log^2(n)} \cdot \frac{1}{B}$. It turns out that the version without predictions needs a logarithmic factor more space, such that $B' = B \log B$.

Lemma 3.2.1

CountSketch++ without predictions using a space budget of $B' = \mathcal{O}(B \log B)$ achieves the same weighted error as its learned counterpart with space budget B, that is:

$$\frac{1}{N} \sum_{i \in [n]} f_i \cdot |\hat{f}_i - f_i| = \mathcal{O}\left(\frac{N}{\log^2(n)} \cdot \frac{1}{B}\right)$$

Proof Intuition Let us recap some insights from earlier sections to intuitively understand why increasing the space budget by a logarithmic factor works. First, the width of the small sketch tables determines which order of magnitude of elements the algorithm can distinguish. This is due to the fact that the error of a standard CountSketch is inversely proportional to the width of the sketch table (cf. [Aam+23], Theorem C.4). Since we only use three hash rows per table, as a constant number of hash rows is asymptotically optimal, we can write the equations in terms of the sketch size B instead of the width w in the following to remain consistent with the notation in other proofs of this thesis:

Given that the size of the small hash tables is $\mathcal{O}(B)$, the algorithm can tell in the filtering step whether an element is significantly more frequent than $\frac{A}{B}$, i.e., if $f_i = \omega(\frac{A}{B})$, or whether the element's frequency is in the order of magnitude $\frac{A}{B}$ or smaller, i.e., $f_i = \mathcal{O}(\frac{A}{B})$.

Suppose now that we allocate a space budget of $B' = \mathcal{O}(B \log B)$ for the big table in CountSketch++. Following the same reasoning it is preferable in this case to use the estimate from the big sketch table rather than outputting zero for all elements $i \leq B \log B$.

With these observations, the proof proceeds exactly as in Section 3.2. Intuitively, the algorithm splits the elements into three groups: Heavy Hitters, very rare elements, and elements in between these two groups that are close to the threshold τ. For the heavy and rare elements, the algorithm will label them correctly in the filtering step with very high probability. For elements with frequencies close to the threshold, the algorithm causes the highest error when outputting zero.

The only thing that changes compared to the analysis in Section 3.2 are the values plugged in for the threshold τ and the space budget B. To match the performance of a learned CountSketch++ using space B with a standard CountSketch++ (i.e., without predictions), choose the following parameters:

1. Threshold to output zero: $\tau = \frac{A \cdot \mathcal{O}(1)}{B}$
2. Total space budget: $B' = \mathcal{O}(B \log B)$

3. Width of the small sketch tables: $w = \mathcal{O}(B)$

4. Width of the big sketch table: $w' = \mathcal{O}(B \log B)$

As only minor changes that concern asymptotic values of variables but not the structure are necessary to the proof in Section 3.2, we omit the full proof at this point. The interested reader can find the complete proof in Appendix A.

3.3. Estimations Using Predictions

The learned version of the CountSketch++ algorithm assumes that the algorithm has access to an oracle that can perfectly predict the top-$\frac{B}{2}$ elements in the stream without mistakes. Given that knowledge, the learned version uses half of the total space budget B to store the exact counts of these elements. The other half of the space budget remains for setting up the same data structures as before in the standard version with half the space. To compensate for the reduced space, the number of small and large tables remains unchanged, but their size, or more precisely the width of the sketch tables, is adjusted to the smaller space.

As touched upon in Section 3.1.2, the learned CountSketch++ algorithm has several weaknesses that we discuss in detail in the following sections.

3.3.1. A Trivial Alternative to Learned CountSketch++

Since the learned CountSketch++ algorithm only uses the predictions to store the top-$\frac{B}{2}$ elements exactly, the error solely comes from the remaining elements i, such that $i \in [n] \setminus [\frac{B}{2}] \Rightarrow i \geq \frac{B}{2} \Rightarrow f_i \leq \frac{2A}{B}$.

If the algorithm works well, the estimate from the small tables should be close to the true element frequencies. More specifically, the estimate should be asymptotically smaller than $\tau = \frac{A\mathcal{O}(\log \log n)}{B}$ for all of these remaining elements that are not predicted to be in the top-$\frac{B}{2}$. This means that if the algorithm works as desired, it should output zero for all elements that are not tracked with exact counters in a separate array (cf. Figure 3.5).

Instead of setting up a data structure for the remaining elements, one could take a shortcut: Only track the top-$\frac{B}{2}$ elements exactly, and return zero for all other elements. This approach leads to the same asymptotic error as the learned CountSketch++ algorithm while using half

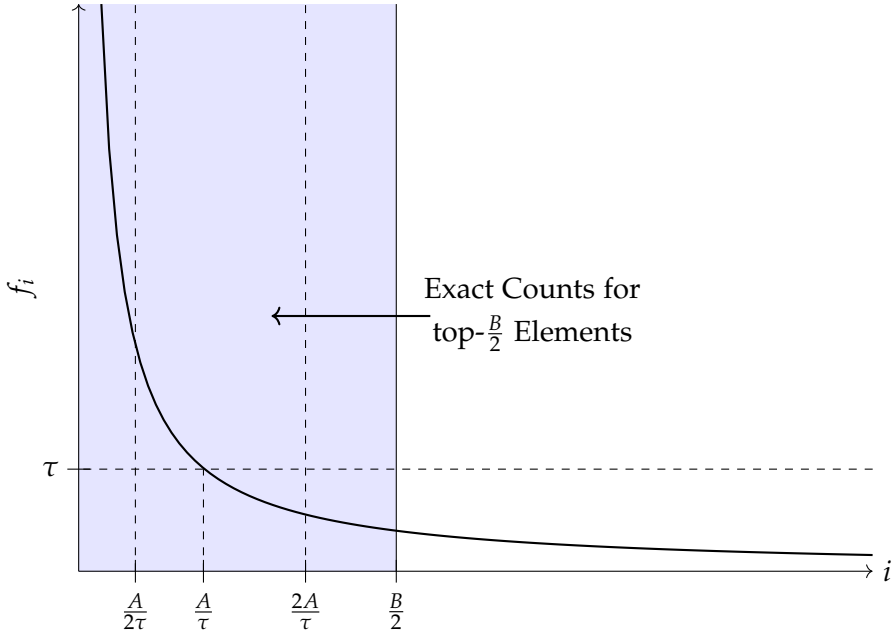


Figure 3.5.: Given that the top-$\frac{B}{2}$ elements are known and tracked exactly, the algorithm should output zero for all remaining elements, since their frequencies are (asymptotically) smaller than τ.

the space:

$$
\begin{aligned}
E_{total} &= \frac{1}{N}\left(\sum_{i=1}^{\lfloor \frac{B}{2} \rfloor} \mathbb{E}[f_i \cdot |\hat{f}_i - f_i|] + \sum_{i=\lfloor \frac{B}{2} \rfloor + 1}^{n} \mathbb{E}[f_i \cdot |\hat{f}_i - 0|]\right) \\
&= \frac{1}{N}\left(0 + \sum_{i=\lfloor \frac{B}{2} \rfloor + 1}^{n} \mathbb{E}[f_i \cdot |\hat{f}_i - 0|]\right) && \textit{Error for top-}\frac{B}{2}\textit{ elements is zero due to exact counts} \\
&\lesssim \frac{1}{N} \sum_{i=\lfloor \frac{B}{2} \rfloor + 1}^{n} \frac{A^2}{i^2} && \textit{Error when outputting zero for all remaining elements is } \frac{A^2}{i^2} \textit{ per element} \\
&\lesssim \frac{A^2}{N} \int_{\lfloor \frac{B}{2} \rfloor + 1}^{n} \frac{1}{x^2}\, dx \lesssim \frac{N^2}{\log(n)^2 N}\left(-\frac{1}{n} + \frac{2}{B}\right) && \textit{Upper bound the sum by the integral} \\
&= \mathcal{O}\left(\frac{N}{\log^2(n)} \cdot \frac{1}{B}\right)
\end{aligned}
$$

□

3.3.2. Feasibility Analysis of the Oracle

The previous section showed that it suffices to output zero for all elements not tracked with exact counts in order to achieve the same asymptotic error as the learned CountSketch++ algorithm. An oracle, as described in [Aam+23], therefore, simplifies the problem considerably, which raises concerns how feasible such a tool is in practice

Furthermore, there are two major concerns with the analysis presented in [Aam+23] with regard to the oracle: First, the use of a perfect oracle that never errs is a strong assumption. Its practicality in real-world settings is unclear and may be overly idealized for some real-world applications. Second, even under this assumption, the analysis in [Aam+23] does not consider the time and space complexity to implement, run, and store such an oracle.

The authors in [Aam+23] suggest to use a neural network as an oracle. As already mentioned, they do not account for the additional time and space complexity required to train and store such a network in their analysis. Using a neural network as an oracle brings several downsides: First, it takes a long time and requires significant computing resources to train it. Second, the training process does not scale well across domains: A neural network trained on internet traffic data is unlikely to perform well on frequent words in an NLP context. Third, successful training requires that sufficient representative data is available. Fourth and lastly, the neural network would only manage to find patterns in the data stream if these actually existed and are stable over time.

These limitations motivate the need to explore more practical and realistic oracles that better align with computational and domain-specific constraints.

3.4. Realistic Oracles

This section discusses how to implement realistic oracles while staying within the space budget B. We start by discussing the two-pass setting. As the name suggests, it allows two passes over the data: one pass to build the oracle and a second pass to run the learned CountSketch++ that queries the oracle built during the first pass to know the top-k elements. In the second part, we explore what is feasible when only one pass over the data is allowed. Throughout all the settings and variations, we consider the following oracles and their respective adaptations:

Counter-Based Oracles	**Sketch-Based Oracles**
• Misra-Gries • SpaceSaving	• CountMin • CountSketch • CountSketch++

Table 3.2.: Examined types of oracles

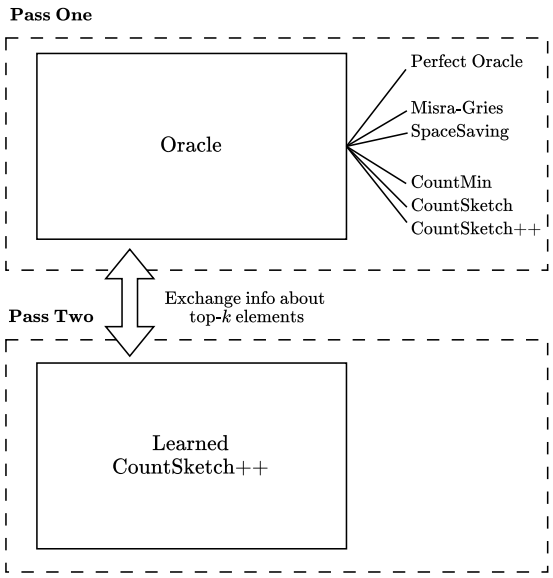


Figure 3.6.: When it is feasible to run two passes over the data, the oracle is built in the first pass and used in the second pass for building the learned CountSketch++.

3.4.1. Two-Pass Setting

In this setting, the first pass is used to build the oracle, and the second pass runs the learned CountSketch++ data structure, which queries the oracle to identify the top-k elements as illustrated in Figure 3.6. Allowing two passes over the data approaches the problem of designing a suitable oracle as the problem of finding an algorithm that can predict the top-k elements in a single pass.

Although this setting requires storing the whole data stream to repeatedly run over it, it is a reasonable assumption in some settings. This includes situations, where the actual frequency estimation should run in memory since it would be too costly to read the data from disk every time an estimate is needed. You can think of this scenario as compressing the frequency statistics of a large dataset into a small sketch that fits into memory. Another example are distributed settings, when it might not be feasable or desirable to send the whole part of the data stream between parties. Again, the frequency estimation algorithms works as a form of compression in this case.

As introduced in Section 2, there are two types of algorithms for finding top-k elements: deterministic counter-based algorithms like Misra-Gries and SpaceSaving, and randomized sketch-based algorithms that rely on hashing.

Since asymptotically it does not make a difference whether one considers the top-B or top-$\frac{B}{2}$ elements, we will refer to $k = B$ instead of $\frac{B}{2}$ in the following to simplify notation.

Counter-Based Oracles Although Misra-Gries and SpaceSaving are from a theoretical point of view intended to find Heavy Hitters, i.e., all elements whose frequency exceeds a certain threshold, it is possible to tweak them to output only k elements. To do so, simply return the elements corresponding to the k highest counters. Whether these are indeed the most frequent elements, depends on the frequency distribution in the data stream and the actual value of k. In the present case, we want to retrieve the top-B elements, therefore $k = B$. According to the theoretical guarantees, it holds that given k counters, all elements that appear more than $\frac{N}{k+1}$ times in the stream will certainly be in the algorithm's output. To recover the top-B elements in a Zipfian data stream, i.e. all elements with frequency larger than $\frac{A}{B}$, Misra-Gries and SpaceSaving would require space:

$$\frac{A}{B} = \frac{N}{B \log n} \geq \frac{N}{k+1} \Rightarrow k = \mathcal{O}(B \log n)$$

I.e., given $k = \mathcal{O}(B \log n)$ words of memory, the algorithms will find the top-B elements in a Zipfian data stream.

Sketch-Based Oracles As described in Section 2.3.3, sketch algorithms initially designed for frequency estimation, can be adapted for finding frequent elements. To do so, the algorithm adds a heap of size k, which tracks the top-k predicted frequencies at any time. When first introducing the mechanism for CountSketch, [CCFC04] found that for a Zipfian distribution with parameter $a \geq 1$, the algorithm needs $\left(k \log \frac{N}{\delta}\right)$ words of memory to find the top-k elements with probability $1 - \delta$. Again, N denotes the total stream length. Similar analyzes hold for CountMin and CountSketch++, when combining them with a heap to find frequent elements. Like the two counter-based approaches discussed in the last paragraph, these algorithms require much more space to reliably find the top-B elements than the available space budget B.

The theoretical analysis above shows that reliably recovering the top-B elements requires more than the desired space budget B. Nevertheless, in practice, a CountSketch++ in a second pass may help correcting some of the oracle's errors by outputting the estimate from the big sketch table instead of returning zero for heavy elements that were not properly recovered as top-B element by the oracle. Using different datasets, Chapter 4 explores to what extent this is possible and how the oracles differ in performance in practice.

3.4.2. One-Pass Setting

Allowing two passes over the data has one central limitation: The whole stream needs to be stored. While this is reasonable and feasible to assume for some settings as explained previously, it might not be suitable for scenarios where minimizing memory usage is a priority. The question, therefore, is how to detect Heavy Hitters and estimate frequencies in one pass.

Using Oracle Counts for Frequency Estimation The most straightforward approach is to omit the second pass and rely solely on the information gathered while building the oracle (cf. Figure 3.7): All oracles considered in this work have some form of counts/estimates associated with the predicted top-k elements. Instead of running a separate frequency estimation in an extra pass, one can simply output the count given by the oracle if an element is in the predicted top k, and set the estimate to zero otherwise. This approach is similar to the trivial

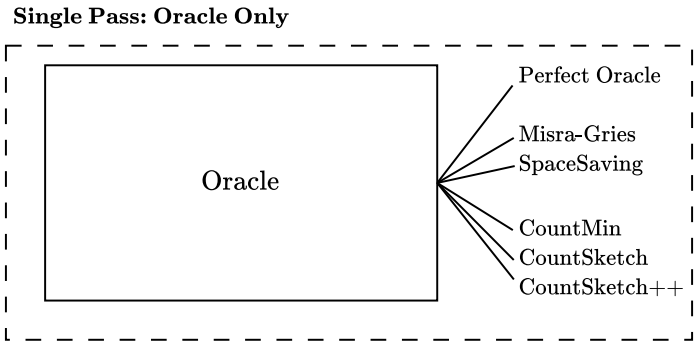


Figure 3.7.: When only a single pass over the data is possible, the most straightforward strategy is to run a Heavy Hitter / top-k algorithm and use its output as frequency estimates for the predicted top-k elements, while assigning zero to all other elements.

alternative discussed in Section 3.3.1. The mathematical analysis there, however, assumes a perfect oracle. As stated, the only upper bounds known for the oracles discussed require significantly more space than the available space budget B to succeed with high probability. The theoretical analysis suggests, therefore, that for some Heavy Hitters, this mechanism will very likely incur an error much larger than $\frac{1}{B}$ when using space $\mathcal{O}(B)$.

The advantage of this approach is that it is space and time efficient: Having only an oracle requires less space and time to update the data structures compared to running an additional data structure in parallel.

Augmented Oracles A way to mitigate the problem of misclassified Heavy Hitters as explained above is to combine the oracles with a CountSketch++ instance: While building the oracle, one can also run a CountSketch++ data structure in parallel (cf. Figure 3.8). We will refer to this approach – an oracle plus a CountSketch++ – as an augmented oracle. The CountSketch++ serves as a backup to correct mistakes made by the oracle. In case the oracle does not detect a Heavy Hitter as such, the CountSketch++ would still label it as frequent upon query time and output the estimate from the big sketch table with very high probability.

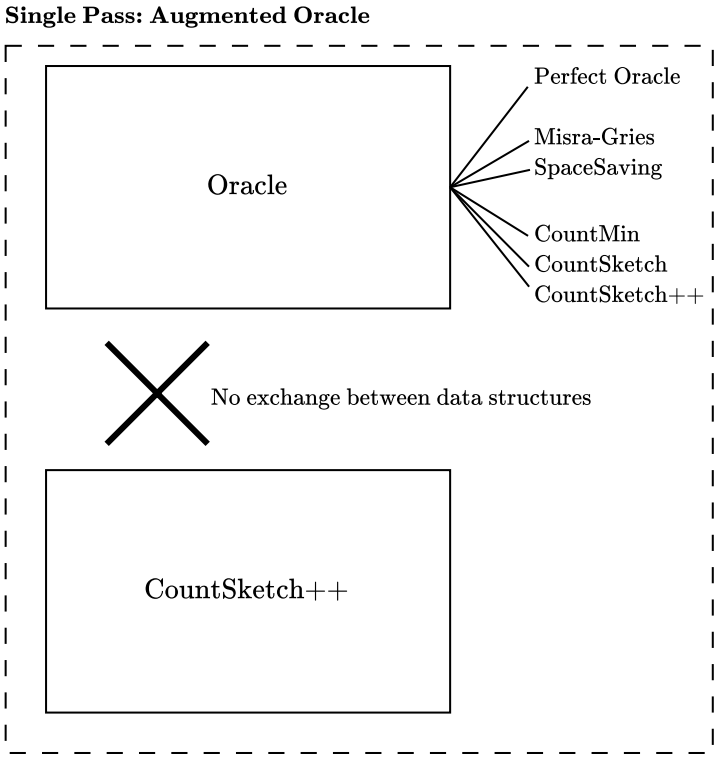
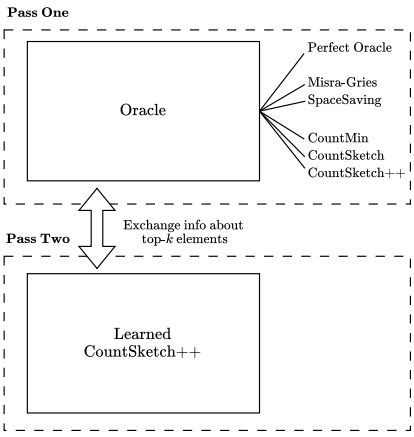


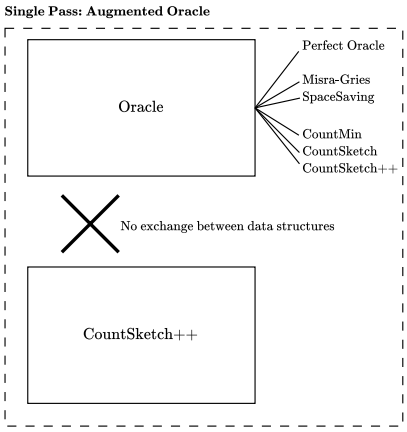
Figure 3.8.: A more refined approach for one-pass settings than using oracle counts directly is to use augmented oracles. These run a separate CountSketch++ instance alongside the oracle itself.

3.4.3. Overview of the Oracles Presented

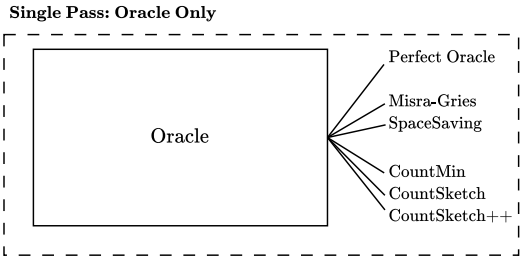
Overall, we presented three kinds of approaches in this chapter to implement realistic oracles. The first approach assumes that two passes are possible. It builds an oracle in the first pass and uses it in the second pass to run the learned CountSketch++ (cf. Figure 3.9a). The second approach only allows one pass and solely relies on the oracle's output for frequency estimation (cf. Figure 3.9c). The third approach also works in one pass but builds an additional CountSketch++ instance alongside the oracle to mitigate some of the oracle's mistakes (cf. Figure 3.9b). We refer to this third approach as augmented oracle. Figure 3.9 summarizes these three approaches.



(a) Two passes: The oracle is built in the first pass and used in the second pass for building the learned CountSketch++.



(b) One pass: An augmented oracle runs a separate CountSketch++ instance alongside the oracle itself.



(c) One pass: Only the oracle is built. The counts associated with the top-k elements serve as frequency estimates.

Figure 3.9.: Different approaches to implement oracles in practice with one and two passes over the data.

4. Experiments

The experiments in this chapter are thematically divided into two parts: The first part of the evaluation considers settings that allow for two passes over the data, while the second part focuses on the more restrictive one-pass scenario. The first experiment in part one, experiment 1.1, examines how the choice of oracle in the first pass affects the performance of CountSketch++ in the second pass. Figure 4.1a illustrates the experiment setup. Experiment 1.2 then varies the behavior at query time as shown in Figure 4.1b: For each oracle, we compare the error obtained when querying the trained CountSketch++ against a trivial baseline that simply outputs zero for all estimates outside the top-$\left(\frac{B}{2}\right)$. The experiments in part two deal with the performance of the oracles in a one-pass setting. Experiment 2.1 compares how well the oracles perform when using their counts as estimates without any further data structure for frequency estimation (cf. Figure 4.1c). Finally, we will investigate the performance of augmented oracles, which combine an oracle with a CountSketch++ instance in experiment 2.2. As shown in Figure 4.1d, the oracle and the CountSketch++ are built in one step and do not exchange any information in this setting unlike the setup for experiment 1.1.

The key insights are:

Experiment 1.1: **Oracles + learned CountSketch++ (2 Passes)**
When combined with a CountSketch++ in the second pass, the Misra-Gries and SpaceSaving oracles perform best among the realistic oracles. They achieve the same performance as perfect oracle for Zipfian data.

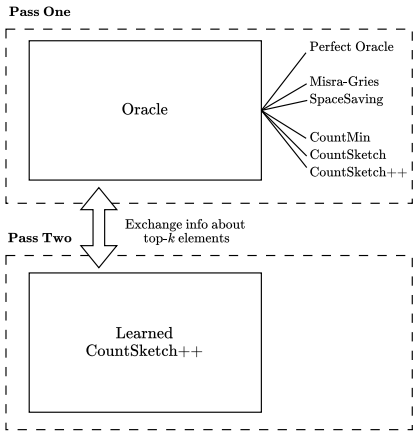
Experiment 1.2: **Oracle + Trivial Mechanism vs. Oracle + learned CountSketch++ (2 Passes)**
Given Zipfian data, the trivial mechanism that outputs zero for all non-Heavy Hitters performs equally well as the learned CountSketch++ algorithm when using counter-based oracles to predict the top-k.

Experiment 2.1: **Oracle Only (1 Pass)**
Using the counts from counter-based oracles as frequency estimates outperforms the standard CountSketch++ on Zipfian data.

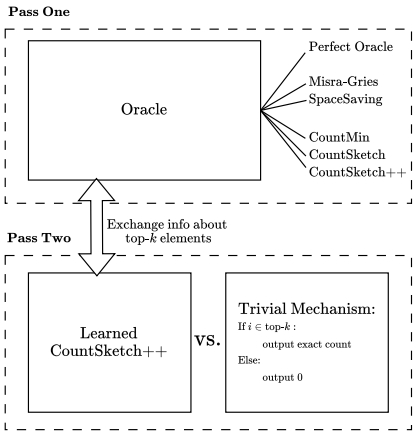
Experiment 2.2: **Augmented Oracles (Oracle + CountSketch++) (1 Pass)**
Augmented counter-based oracles that combine traditional oracles with CountSketch++ instances perform on par with the augmented perfect oracle.

Overall Comparison:
For Zipfian data, using the counts from the SpaceSaving oracle as frequency estimates is just as good as any of the two-pass approaches, including approaches that assume a perfect oracle.

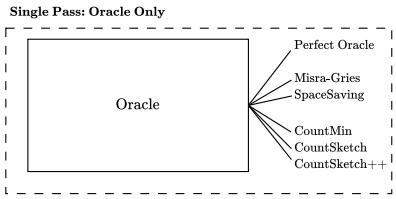
Experimental Setup We ran all experiments for every oracle listed in table 3.2 five times with different random seeds and report the average results and the standard deviation. In addition,



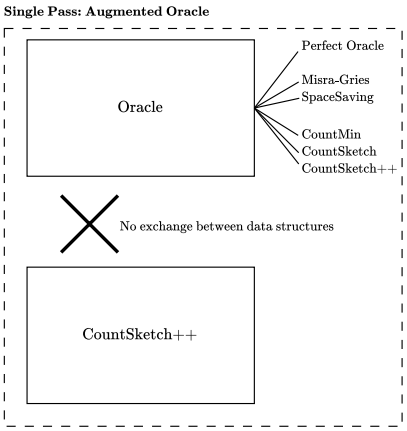
(a) Experiment 1.1: Oracles + learned CountSketch++ (2 Passes)



(b) Experiment 1.2: Oracles + Trivial Mechanism vs. Oracles + learned CountSketch++ (2 Passes)



(c) Experiment 2.1: Oracle Only (1 Pass)



(d) Experiment 2.2: Augmented Oracles (1 Pass)

Figure 4.1.: Overview of experiment setups

we varied the threshold parameter τ, which decides whether CountSketch++ outputs zero or instead returns the estimate from the large sketch table, since the theoretical analysis defines asymptotic values for τ:

$$\tau = \frac{\mathcal{O}(\log\log n)}{B} = \frac{C \cdot \log\log n}{B}$$

We tested three different values for the factor C in the asymptotic bound of τ. Unless the results differ significantly based on the choice of C, we will only report the results for one of the constants tested. We provide the results for all constants in Appendix B.

The experiments were conducted on an Apple Silicon M1 chip with 32 GB of memory. To reproduce the experiments, we provide the code via this repository. We are not permitted to share the real-world datasets used in the experiments via the repository. For further information and detailed instructions, please refer to the README file in the repository.

Datasets We compare the different variations of the oracles on the same three datasets as in the original CountSketch++ paper [Aam+23]. Two of them – AOL [PCT06] and CAIDA [CAI19] – are real-world datasets. The third one is a synthetically generated Zipfian data stream. The AOL dataset [PCT06] contains internet search queries over 80 days. There are 2.783.642 queries in total, spread over 1.244.494 unique queries. For one typical day, the data comprises $\approx 3 \cdot 10^4$ total queries and $\approx 19^4$ unique queries. For our experiments, we use the data from the 74^{th} day of the recording[1]. This contains 30.709 total queries and 19.727 unique queries, which is close to the median of the dataset for both values.

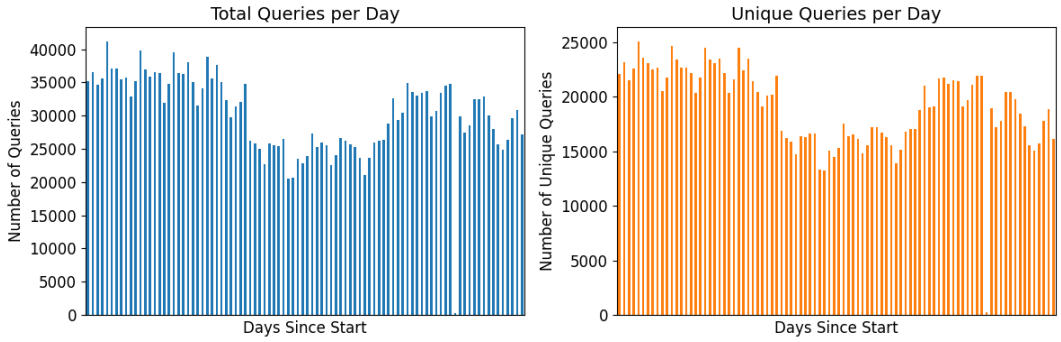


Figure 4.2.: Total and unique queries per day in the AOL dataset

The CAIDA dataset collection [CAI19] provides anonymized internet traffic traces captured using the high-speed equinix-nyc monitor. Each month, the Center for Applied Internet Data Analysis (CAIDA) collects one hour of traffic data.[2] We ran our experiments on the first minute of the recordings from January 2019. The recorded minute contains 29.922.875 Tier1 ISP packets from 326.358 different source IP addresses.

Both datasets exhibit skewed distributions as shown in Figure 4.3.

[1]The 74^{th} day of the recording corresponds to the day with index 73 in the dataset, as the indexing starts at zero.
[2]Since 2014, CAIDA only releases one trace per quarter due to storage constraints.

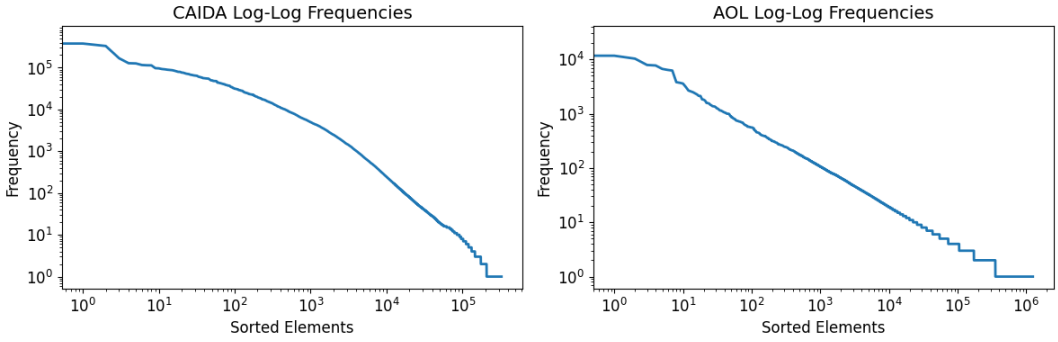


Figure 4.3.: Log-Log plots of the sorted frequencies of items in the CAIDA and AOL datasets.

For the synthetic data, we generate a Zipfian data stream with $n = 5 \cdot 10^5$ such that the i^{th} element has a frequency of $f_i = \lceil \frac{n}{i} \rceil$. We round up the term to ensure every element appears at least once.

4.1. Results and Discussion

To ensure a fair comparison, we allocate the same amount of space B to all oracles during training. This means that heap-based oracles like Misra-Gries and SpaceSaving use B counters. To get the top-k elements in the end, we sort the counters and take the k largest ones. For the sketch-based oracles, we divide the space B between the sketch and the heap, such that there are as many counters as the number of predicted elements k used later in the experiments.

4.1.1. Part 1: Two-Pass Setting

Part one of the experiments evaluates the quality of different oracles in a two-pass setting. During the first pass over the data, the oracles build their internal data structures to make predictions. The second pass uses these predictions to run the learned CountSketch++ algorithm that stores exact counts for the predicted top-k elements.

This part contains two different analyzes: First, we compare how the choice of the oracle influences the performance of the learned CountSketch++. We include the standard CountSketch++ algorithm, i.e. CountSketch++ without any oracle, as a baseline in the comparison as well as the perfect oracle. The second analysis compares for each oracle, how the error differs between using the learned CountSketch++ algorithm in the second pass versus the trivial mechanism that outputs zero for all non-Heavy Hitters.

Experiment 1.1: Realistic Oracles vs Perfect Oracle

This experiment builds different oracles in the first run and uses their predicted top-k elements in the second pass to run the CountSketch++ algorithm. We compare the results of the implemented oracles to the perfect oracle and additionally to the standard CountSketch++ algorithm (i.e. without oracle) as a baseline in the comparison.

Key Insight The theoretical analyzes of the different oracles suggest that none of them is guaranteed to find the top-k elements within space budget $\mathcal{O}(B)$. As this experiment suggests the counter-based oracles still succeed in recovering many of the top-k elements correctly.

> *When combining with a CountSketch++ in the second pass, the Misra-Gries and SpaceSaving oracles perform best among the realistic oracles. They achieve the same performance as the perfect oracle for Zipfian data.*

Results As shown in Figure 4.4, the results differ between the datasets. For the synthetic and the CAIDA dataset, the counter-based oracles Misra-Gries and SpaceSaving perform on par with the perfect oracle. In fact, given space $B = 665$ for the CAIDA dataset and space $B = 224$ for the synthetic dataset, the counter-based oracles return the correct top-k elements. Sketch-based oracles, however, perform significantly worse. Combining them with the learned CountSketch++ algorithm in the second pass does not perform better than the standard CountSketch++ algorithm that does not use any predictions. For the approaches using counter-based oracles, the performance only increases up to a certain space budget. For combinations of a sketch-based oracles and learned CountSketch++, like for the CountSketch++ only without any oracle, the performance improves with increasing space for the CAIDA and the synthetic dataset.

For the AOL dataset, all oracles perform poorly. In fact, they do not improve the performance of the CountSketch++ algorithm over its standard counterpart at all. Even with increasing space, this does not change, and performance remains poor for all oracles.

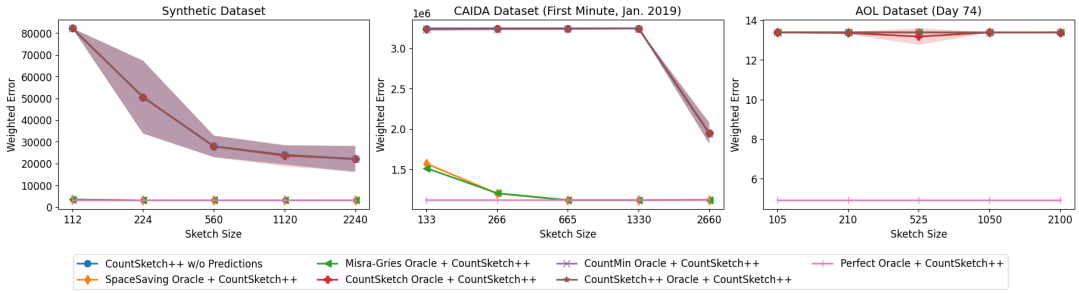


Figure 4.4.: Comparison of weighted error on all three tested datasets for different oracles in the first pass. All approaches compared use the CountSketch++ algorithm (C=8) in the second pass, except for the standard CountSketch++ which serves as a baseline.

Discussion Although no theoretical bounds are known as of now that guarantee that any of the oracles can approximate the top-B elements within space $\mathcal{O}(B)$ sufficiently well, the counter-based oracles recover them nearly completely correctly for the synthetic and the CAIDA data in practice. Although Misra-Gries and SpaceSaving both produce some errors for the experiments with the smallest space budgets, these errors only involve mistaking top-k elements with other very heavy elements. Furthermore, the CountSketch++ algorithm in the second pass seems to be able to detect the wrongly labeled top-k elements in the filtering step

as such and mitigates these misclassifications by outputting the estimate from the big sketch table instead of returning zero.

The observation that the error for the unlearned CountSketch++ declines at a similar rate as the sketch-based oracles for the synthetic and the CAIDA data suggests that it is not the oracles that improve with increased space. In fact, the increase in space does not affect the quality of their predictions. Rather, the CountSketch++ algorithm delivers better estimate with increasing space and, hence, lowers the error.

For the AOL dataset, all the oracles fail to find or approximate the top-k elements well enough. To better understand why the results differ so much between the datasets, it is helpful to look at their distributions again. When comparing how skewed the AOL dataset is compared to a perfectly Zipfian dataset, we see that the distribution of the AOL dataset is much flatter than the synthetic dataset (cf. Figure 4.5). The Heavy Hitters in the AOL dataset have a much lower relative frequency than expected for a Zipfian distributed dataset, unlike the CAIDA dataset (cf. Figure 4.6). I.e., the Heavy Hitters in the AOL data stand out less from the rest of the data. This makes it harder for the oracles to detect and keep them in their heaps. In the Misra-Gries algorithm, for example, the counter for an element in the heap is decremented nearly each time an untracked element arrives. Consequently, a Heavy Hitter only remains in the heap if it survives all the decrements incurred by rare elements. The smaller the difference between Heavy Hitters and the rest of the data, the more likely it is that the rare elements decrement the count of a Heavy Hitter to zero and throw it out of the heap. We therefore conclude that the oracles require strongly skewed data to work efficiently. We leave it for future work to quantify the level of skewness required for the oracles to work well in practice.

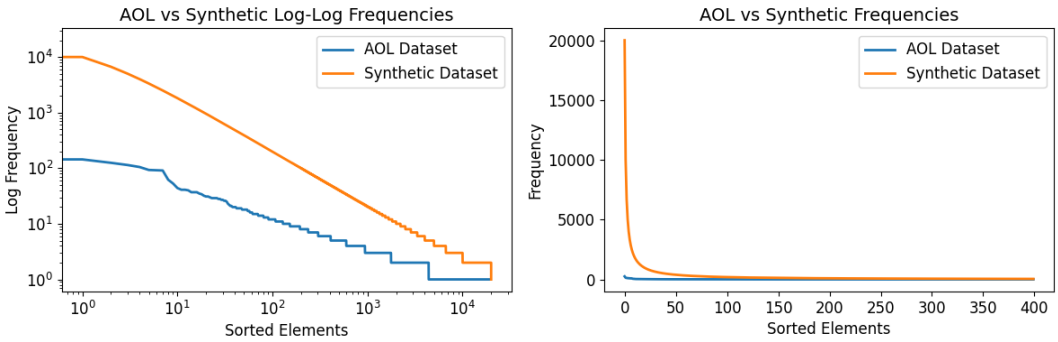


Figure 4.5.: Compared to the synthetic dataset the AOL dataset is less skewed.

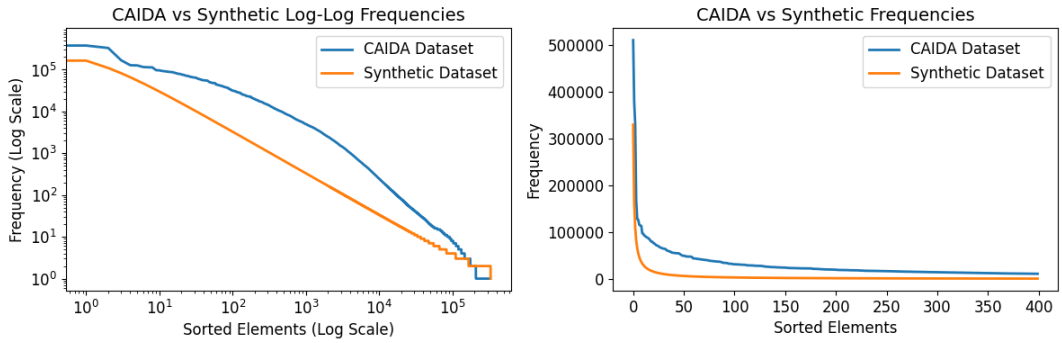


Figure 4.6.: The CAIDA dataset is strongly skewed. Heavy Hitters are significantly more frequent than the rarer elements in the dataset.

Experiment 1.2: Learned CountSketch++ vs Trivial Mechanism

Experiment 1.1 has shown how varying the oracles as the first building block of the two-pass setting affects the performance of the learned CountSketch++ algorithm in the second pass. Given that insight, the question emerges: How much does the second building block, the learned CountSketch++ algorithm itself, contribute to the performance? Therefore, the experiment 1.2 investigates to what extent the theoretical results from Section 3.3.1 hold in practice. Given different oracles trained during the first pass, the experiment explores to what extent the error differs between the trivial mechanism versus running the learned CountSketch++ algorithm in the second pass. The trivial mechanism as described in 3.3.1 only keeps exact counts for the predicted top-k elements but does not keep an additional sketch table for the remaining elements. Instead, it sets the estimate to zero during query time for these elements.

Key Insight The analysis in Section 3.3.1 proves that from a theoretical point of view, outputting zero for all elements that are not predicted to be in the top-k achieves the same asymptotic error as the learned CountSketch++ algorithm (i.e., oracle + learned CountSketch++). This experiment suggests that this holds true in practice even for some of the non-perfect oracles.

Given Zipfian data, the trivial mechanism that outputs zero for all non-Heavy Hitters performs equally well as the learned CountSketch++ algorithm when using counter-based oracles to predict the top-k.

Results Figure 4.7 shows the performance on the CAIDA data of the CountSketch++ oracle, the Misra-Gries, and the perfect oracle when combined with a CountSketch++ in the second pass versus the trivial mechanism that outputs zero for all non-top-k elements. The SpaceSaving oracle yields similar results to the Misra-Gries oracle, whereas the results for the CountSketch++ oracle are representative of all other sketch-based mechanisms. You can find the results for all oracles in Appendix B.
The results for the synthetic dataset (cf. Figure 4.8) and the AOL dataset (cf. Figure 4.9) follow similar patterns.

Only combined with the sketch-based oracles, using the CountSketch++ algorithm in the second pass yields an improvement compared to outputting zero for non-Heavy Hitters for the CAIDA data. This effect is even more pronounced for the synthetic dataset that follows a perfect Zipfian distribution. The error, however, is still significantly higher than that of the other oracles. For the qualitatively better counter-based oracles, outputting zero always yields at least as good results as using a sketch in the second pass.

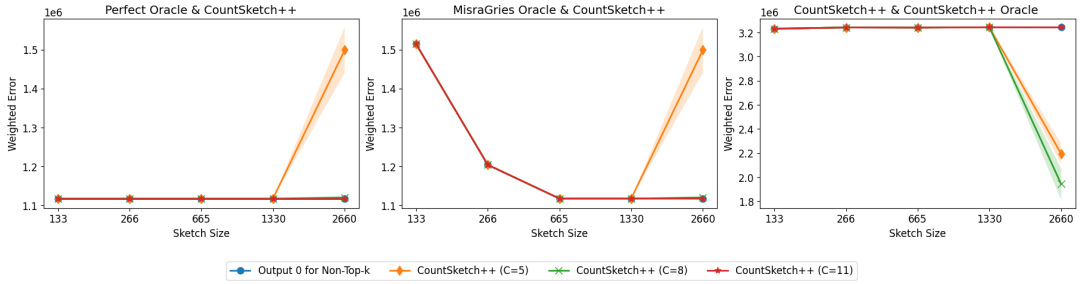


Figure 4.7.: Comparison of the weighted error on the CAIDA dataset for different oracles in the first pass combined with a learned CountSketch++ vs. outputting zero for all non-top-k elements.

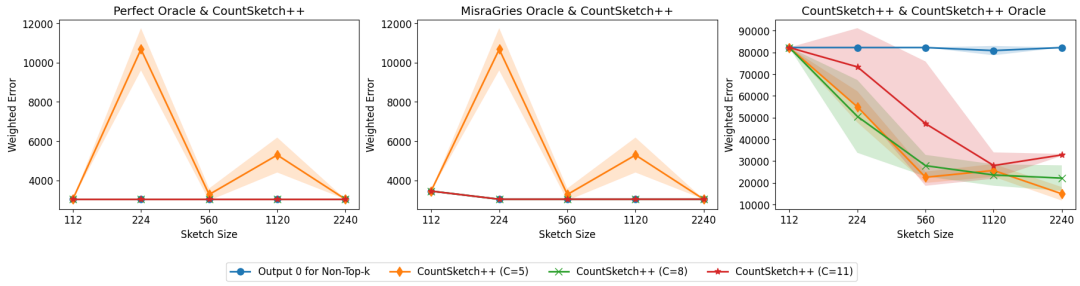


Figure 4.8.: Comparison of the weighted error on the Synthetic dataset for different oracles in the first pass combined with a learned CountSketch++ vs. outputting zero for all non-top-k elements.

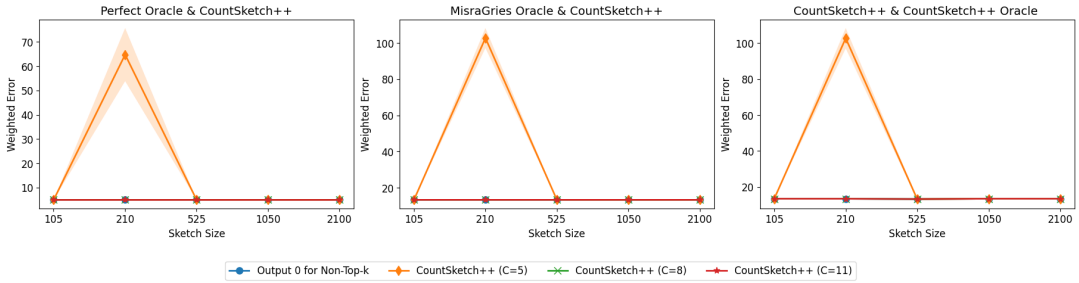


Figure 4.9.: Comparison of the weighted error on the AOL dataset for different oracles in the first pass combined with a learned CountSketch++ vs. outputting zero for all non-top-k elements.

Discussion The practical experiments confirm the theoretical results from Section 3.3.1: Given a qualitatively meaningful oracle, i.e., one that only makes minor errors, it does not matter whether to run the learned CountSketch++ algorithm or not in a second pass over the data. It suffices to track the top-k elements and output zero for all other elements. Only when the oracles can not predict reasonable estimates for the top-k elements, as is the case for sketch-based oracles, it is better to run the learned CountSketch++ algorithm in the second pass compared to the trivial mechanism. This is because the sketch in the second run is able to mitigate errors made by the oracle as explained earlier.

As the previous experiment has shown, all of the realistic, i.e., non-perfect oracles fail to find meaningful Heavy Hitters due to the comparably flat distribution of the AOL dataset. Therefore, as with the CAIDA dataset, one would assume that running a CountSketch++ in the second pass will help to detect mislabeled elements and recover the error to some extent. However, the results show that this is not the case. For the AOL data in particular, it does not seem to matter whether the learned CountSketch++ algorithm runs in the second pass or not for any of the oracles. Looking at the mathematical formulation of the error provides a reasonable explanation: Only if an element is heavy, the error term is multiplied by a high weight. If an element is not heavy or even very rare, the weight is small, and the absolute error term therefore contributes little to the overall error. Since the AOL dataset is only moderately skewed, even the weights for the Heavy Hitters are not very high. It therefore does not make too much of a difference on which of the elements the algorithm makes an error, unless it recovers good estimates by keeping exact counts in the second pass for some of the elements. Even when outputting zero for all elements, the weighted error for the AOL dataset is 8.06, whereas for the perfect oracle, the weighted error is 4.91. For comparison, the weighted errors for the CAIDA and the synthetic dataset are both in the six-digit range, even when using a perfect oracle. The reason for the low error for the AOL data is therefore not the quality of the algorithm but rather the distribution of the data.

4.1.2. Part 2: One-Pass Setting

The first part of the experiments considered approaches that require two passes over the data. As discussed in Section 3.4.1, this approach is particularly useful when the data can be stored and processed multiple times, and the main objective is to obtain a compact representation of the frequencies. However, in many scenarios storing the entire data stream is not feasible. Therefore, this second part of the experiments evaluates the suggested methods for a one-pass

setting.

The first experiment of part two, experiment 2.1, examines whether the information the oracles gather in one pass is sufficient to make good estimates. The second experiment, experiment 2.2, explores how to further improve these estimates by leveraging additional information.

Experiment 2.1: Oracles as Frequency Estimators

For the one-pass setting we start with the easiest and most straight-forward approach: We simply omit the second pass. I.e., this experiment uses the counts of the oracles as estimates for the frequencies of the elements without any additional processing. If an element is not predicted to be in the top-k, the algorithm sets its estimate to zero.

Key Insight From a theoretical point of view, the counter-based oracles can output estimates that deviate from the true frequency by at most $\frac{N}{k+1}$, where N is the stream length. This error term is quite significant for our settings, since k is usually much smaller than the total stream length N. From the experiments we see that for Zipfian datasets, the counter-based oracles come close to the performance of a perfect oracle and sometimes even match its performance. They are able to recover the true frequencies of the predicted top-k elements nearly perfectly, and beat the unlearned CountSketch++ algorithm.

Using the counts from counter-based oracles as frequency estimates outperforms the standard CountSketch++ without predictions on Zipfian data.

Results Overall, the counter-based oracles perform best among all realistic (non-perfect) oracles in this experiment. As Figure 4.10 shows, their estimates on their own beat the CountSketch++ algorithm without predictions for the CAIDA and the synthetic dataset. Since the high error for the sketch-based oracles distorts the plot, we omit them for the CAIDA dataset at this point to enable a better understanding of the performance of the counter-based oracles. You can find the plot that includes all oracles in the appendix. SpaceSaving performs slightly better than Misra-Gries and is even able to achieve the same performance as the perfect oracle from a specific space threshold on. This threshold is slightly later for the CAIDA dataset than for the synthetic dataset. The sketch-based oracles, on the other hand, perform poorly. Their estimates are significantly worse than the unlearned CountSketch++ algorithm. For the AOL data, none of the oracles is able to recover meaningful frequencies of the top-k elements. However, the weighted error is comparable to that of the unlearned CountSketch++ algorithm.

Discussion The results of this experiment show that the counter-based oracles can recover the true underlying frequencies of the predicted top-k elements nearly perfectly for very skewed data streams. Although the counter-based oracles are no more accurate in identifying the Heavy Hitters with increasing space, the increased space budget improves the precision of the frequency counts for the predicted top-k elements. This lowers the error and makes the SpaceSaving algorithm competitive with the perfect oracle.

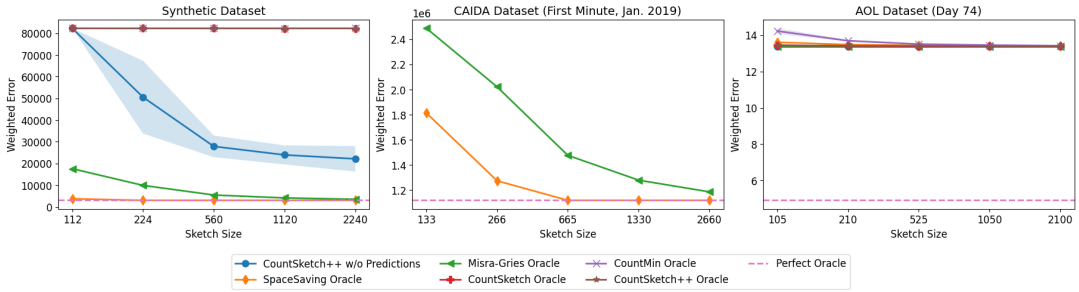


Figure 4.10.: Comparison of the weighted error when using the counts produced by the oracles in one run as frequency estimates.

An intuitive explanation why SpaceSaving outperforms Misra-Gries is that even if the top-k elements stay in the heap from their first appearance till the end of the stream, Misra-Gries decrements their counts every time an untracked element arrives and the heap is already completely occupied. I.e., untracked elements in Misra-Gries almost certainly affect the counter of the top-k elements at some point. For SpaceSaving, this is not necessarily the case. An untracked element only changes the count of at most one element in the heap, namely, the current minimum count. As long as a given top-k element enters the heap while it is not full yet and is never the element with the minimum counter in the heap, SpaceSaving, therefore, recovers its estimate exactly. Intuitively, this leads to more precise counts in the SpaceSaving algorithm compared to Misra-Gries.

For the AOL dataset, the oracles can not recover meaningful frequencies. This observation aligns with the previous experiments, as they show that this dataset is not skewed enough for the oracles to find the top-k elements. Following the reasoning from previous experiments, the comparably flat distribution leads to smaller weights, which is why the weighted error is comparable to the standard CountSketch++ algorithm. However, this does not mean the estimates are meaningful, as elaborated in previous sections. Rather, it means that a random algorithm can produce a reasonable weighted error, given the comparably flat distribution of the AOL dataset.

Experiment 2.2: Augmented Oracles

Solely using information from an oracle in a single pass as shown in the previous experiment 2.1 is straightforward, requires little storage, and is time-efficient, as the algorithm only needs to maintain one data structure. The remaining question is how much additional benefit a more refined approach can yield. Therefore, this section augments the oracles with a CountSketch++ instance: While building the oracle, we also build a CountSketch++ that keeps track of the frequencies of all elements in the stream. The two data structures are entirely independent of each other and do not exchange any information. For estimating frequencies, the algorithm in this experiment outputs the estimate from the CountSketch++ instance for all elements that are not predicted to be in the top-k. For the predicted top-k elements, it outputs the estimate from the oracle, like in the previous experiment.

Key Insight The theoretical assumptions and practical insights from previous experiments come together in this experiment. As seen in experiment 1.1, counter-based oracles recover the

top-k elements nearly perfectly, even for large streams like the CAIDA dataset. Experiment 1.2 shows that outputting zero is just as good as running the learned CountSketch++ algorithm in the second pass for a qualitatively meaningful oracle. Lastly, experiment 2.1 shows that not only can the counter-based oracles recover the top-k elements reasonably well, but also that the counts associated with the predicted top-k are close to their true frequencies. Experiment 2.2 combines these insights and shows that, also for their augmented version, counter-based oracles outperform the unlearned version of CountSketch++ significantly for Zipfian data and achieve the performance of the perfect oracle.

Augmented counter-based oracles that combine traditional oracles with CountSketch++ instances perform on par with the augmented perfect oracle.

Results Overall, experiment 2.2 exhibits the same patterns as seen in previous experiments. Again, for the AOL dataset, all the approaches – realistic oracles and the standard CountSketch++ without predictions – produce similar results. For the synthetic and CAIDA datasets, the augmented counter-based oracles perform significantly better than sketch-based oracles and achieve the performance of the perfect oracle for increasing space budgets.

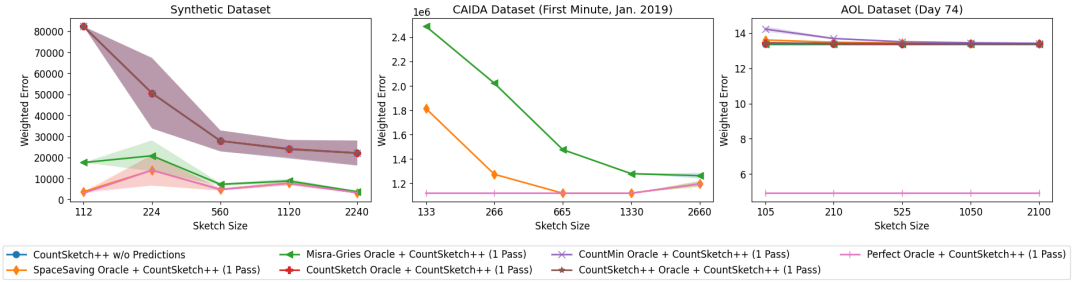


Figure 4.11.: Comparison of the weighted error for augmented oracles on different datasets. To improve readability, we omitted sketch-based oracles in the plot for the CAIDA dataset since their high errors distort the plot. The CountSketch++ uses C=8 for the threshold τ

Discussion To explain the results for this experiment, the same reasoning holds as in the previous section: The distribution in the AOL dataset is too flat for any of the oracles to recover meaningful top-k predictions, which is why it does not make a difference which of the oracles to choose. The performance completely depends on the CountSketch++ instance that runs alongside the oracle. More specifically, it does not matter for the performance whether to have a separate oracle at all for the AOL dataset.

Due to the internal mechanism described in the previous discussions section for experiment 2.2 (cf. Section 4.1.2), the Misra-Gries algorithm performs slightly worse than SpaceSaving for the more skewed CAIDA and synthetic dataset.

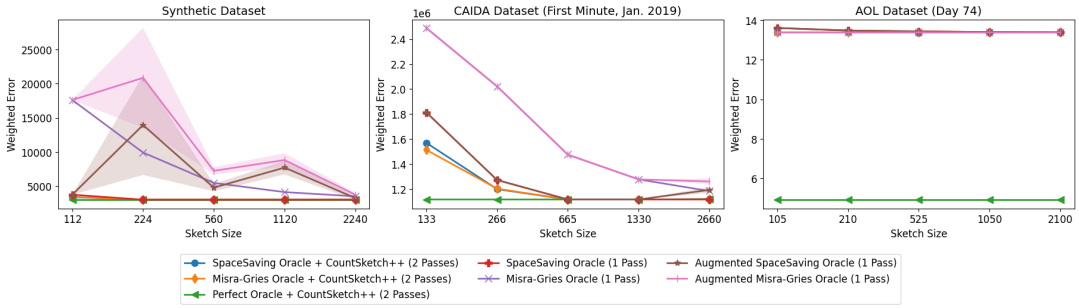


Figure 4.12.: Comparison of the two best performing approaches from each of the previous experiments.

4.2. Overall Comparison

So far, we have evaluated the approaches for two-pass and one-pass settings in isolation and did not compare performance across settings with different number of passes. This section aims to fill this gap and shows how the one- and two-pass setting compare against each other. Bringing it all together, we compare the best two approaches from each experiment across settings. This enables to identify the overall best-performing approach and to see whether having two passes over the data is worth the additional resources.

Throughout all experiments, the best performing approaches where those, that involve the Misra-Gries and SpaceSaving algorithm. Additionally, we include the learned CountSketch++ with a perfect oracle to compare against, as suggested by [Aam+23].

> For Zipfian data, using the counts from the SpaceSaving oracle as frequency estimates is just as good as any of the two-pass approaches, including approaches assuming a perfect oracle.

Using the counts from the SpaceSaving oracle achieves the same performance as all two-pass approaches on the synthetic and the CAIDA dataset. This includes those algorithms that assume a perfect oracle as shown in Figure 4.12. For neither the Misra-Gries nor the SpaceSaving-based approaches in the one-pass setting it offers an advantage to augment them with an additional CountSketch++. Quite to the contrary: An additional CountSketch++, requires setting the threshold parameter τ which might influence the results negatively. Additionally, the CountSketch++ can worsen the overall results if hash collisions occur and a rare element is mistaken for a Heavy Hitter. As long as the data distribution allows the oracles to make meaningful predictions, it is, therefore, preferable to omit the CountSketch++ and rely solely on the counter-based oracles. Since these oracles are deterministic, one does not have to worry about hash collisions. The same input stream will always yield the same outcome. The randomness introduced by the CountSketch++ does not seem to improve the performance but introduces risk for failure. Additionally, the approach of using counter-based oracles only also does not require choosing the threshold parameter τ that might influence results.

Another advantage of using the counts of the SpaceSaving algorithm as predictions is that

it only requires one pass over the data, which is an advantage over approaches that assume that two passes over the data are possible: First, it reduces the overall processing time and resource consumption, making it more efficient for large datasets. Second, it works for more settings, including those where only a single pass is feasible.

Note that [Aam+23] already theoretically proved and showed in experiments on all datasets used in this work, that CountSketch++ outperforms CountMin and CountSketch both in the learned and unlearned setting. Solely using the counts from a SpaceSaving algorithm as estimates for top-k and outputting zero otherwise, therefore outperforms the state-of-the-art approaches for frequency estimation on strongly skewed data in practice.

As we already elaborated, the AOL dataset is not skewed enough for the oracles to work, since the most frequent elements do not stand out enough in the data stream compared to the rest of the elements. It therefore does not surprise that a perfect oracle works significantly better than all other approaches tested. However, the failure of the realistic and implementable oracles also shows how strong an assumption the perfect oracle is for such cases. Even more so, since we included the state-of-the-art algorithms for finding Heavy Hitters / top-k Misra-Gries and SpaceSaving in our experiment. To our knowledge, there are currently no algorithms known that have a better theoretical or practical performance than these two. The perfect oracle's supremacy is therefore of little practical power, since it is still open how to achieve such performance within the given space budget.

4.3. Summary of Experiments

The findings from the experiments suggest that using the counts from the SpaceSaving algorithm as frequency estimates is a strong approach for frequency estimation on Zipfian data. It performs on par with the best two-pass approaches, including those assuming a perfect oracle, while only requiring one pass over the data.

This makes it a resource-efficient and effective choice for frequency estimation in many scenarios not only due to its memory efficiency but also regarding query latency: Estimating frequencies with SpaceSaving is very fast, as it only requires a single hash table lookup and therefore also beats the query-time optimized version of CountMin as studied in [RKA16] with regard to memory consumption, estimation accuracy and query latency.

As the experiments for the AOL dataset show, the approach is tailored to Zipfian data. It is therefore important to ensure that the data stream in question is strongly skewed for the SpaceSaving algorithm to perform well on frequency estimation tasks.

5. Future Work

We organize our suggestions into six sections, each outlining a potential direction for future research. These directions include extending the experimental evaluation to a broader range of parameters and developing supplementary theoretical analyzes to complement practical observations. Additional avenues for exploration involve generalizing the results to a wider variety of query and input distributions.

Systematic Tuning of Space Budget B in CountSketch++: In this work, the space budget was selected to match prior experiments. Future research could develop more principled methods for tuning the concrete value of the space parameter B to give informed guidelines for practical applications.

Parameter Optimization for τ in CountSketch++: Like the space budget, we evaluated the CountSketch++ on a limited set of values for the threshold parameter τ. A more systematic analysis should investigate how different values of τ impact the algorithm's performance across various datasets and query distributions. Like the space parameter B, this could lead to more principled choices for τ, potentially parameterized by the stream length N and the number of distinct elements n.

Heap Size Allocation in Sketch-Based Oracles: In the current design, heap sizes in the sketch-based oracles were fixed to k. The algorithm allocates the remaining space to the hash tables. Future work could explore how varying this ratio between space for the heap and the hash tables impacts performance for sketch-based oracles. Whether a different allocation strategy could improve accuracy to make the sketch-based oracles more competitive remains an open question.

Improved Theoretical Analysis for SpaceSaving: While the SpaceSaving oracle performs exceptionally well in practice, there are, to our knowledge, no theoretical guarantees yet that support this finding. A key avenue for future work is to derive tighter bounds to formally guarantee the effectiveness of SpaceSaving counts for frequency estimation observed in our experiments.

Decoupling Query and Input Distributions: The experiments and the theoretical analysis in this work assume that the query distribution equals the input distribution of the stream. Future work should explore scenarios where this assumption does not hold and the query distribution differs from the input distribution. Studying how algorithms generalize under such distributional shifts could provide a more complete picture of their practical applicability.

Understanding Weighted vs Unweighted Error: In our broader analysis, we also evaluated the unweighted error for all approaches, which we include in Appendix B. For some settings,

there are substantial differences between these two error metrics. So far, we have not investigated the source of these discrepancies and whether they follow a specific pattern. Whether these differences stem from a few high-frequency elements or many low-frequency ones is still an open question and could inform better algorithm design and evaluation strategies.

6. Conclusions

In this thesis, we build upon the design and evaluation of the CountSketch++ algorithm as suggested in [Aam+23]. We provided new insights into the algorithms' working mechanisms, their shortcomings, and proposed alternatives to address these issues.

In the setting without predictions, we refined the analysis of the theoretical upper bounds for the weighted error. Moreover, we demonstrated—both through theoretical proofs and empirical experiments—that employing a learned CountSketch, i.e., CountSketch++ combined with exact counters for the predicted top-k elements as proposed in [Aam+23], offers no advantage over the trivial mechanism. The trivial mechanism maintains exact counts for the top-k elements and assigns zero to all others. This conclusion holds not only under the assumption of perfect oracles that recover the top-k elements correctly, but also when using counter-based oracles such as Misra–Gries and SpaceSaving.

Although the known theoretical analyzes do not guarantee that these algorithms work within the small space budget as defined in our work, our experiments show that they perform well on real-world datasets. Especially, the SpaceSaving-oracle stands out as the best-performing oracle. In fact, using the counts accumulated while building the SpaceSaving oracle as estimates for the top-k elements and outputting zero for the remaining elements, recovers the same weighted error as the learned version of the CountSketch++ that requires two passes and assumes a perfect oracle.

A. Supplementary Proof

In the following, we provide the full proof of Lemma A.0.1 as stated in Section 3. It follows the same structure as the proof of Theorem 3.2.1 in Section 3.2, with only minor changes to the variables.

We restate the lemma here for convenience:

Lemma 3.2.1

CountSketch++ without predictions using a space budget of $B' = \mathcal{O}(B \log B)$ achieves the same weighted error as its learned counterpart with space budget B, that is:

$$\frac{1}{N} \sum_{i \in [n]} f_i \cdot |\hat{f}_i - f_i| = \mathcal{O}\left(\frac{N}{\log^2(n)} \cdot \frac{1}{B} \right)$$

Remember the intuition presented in Chapter 3: Intuitively, the algorithm partitions the elements into three groups (cf. Figure 3.3): Group one contains the heaviest elements. Group two comprises the rarest elements. Finally, group three holds all elements in between whose frequencies are close to the threshold τ used to determine whether an element is heavy or not during the filtering step. We show, that the algorithm outputs the estimate from the big sketch for the heaviest elements (i.e. group one) and outputs zero for the rarest elements (i.e. group two) with high probability.

To match the performance of a learned CountSketch++ using space B with a standard CountSketch++ (i.e., without predictions), choose the following parameters:

1. Threshold to output zero: $\tau = \frac{A \cdot \mathcal{O}(1)}{B}$
2. Total space budget: $B' = \mathcal{O}(B \log B)$,
3. Width of the small sketch tables: $w = \mathcal{O}(B)$,
4. Width of the big sketch table: $w' = \mathcal{O}(B \log B)$.

We are going to use the following theorem as stated and proofed in [Aam+23]:

Theorem A.0.1 *Source: [Aam+23]*

Let $\hat{f}_i$ be the estimate of the i^{th} frequency given by a 3 x $\frac{B'}{3}$ CountSketch table. There exists a universal constant C such that the following inequality holds:

$$\forall t \geq 3/B', \qquad Pr[|f_i - \hat{f}_i| \geq t] \leq C' \left(\frac{\log(tB')}{tB'} \right)^2$$

First, we decompose the total expected weighted error, $\mathbf{E_{total}}$ in three parts. Each part represents the error incurred by the respective group of elements, where group one contains heavy hitters, group two represents rare elements, and group three the elements in between (cf. Figure 3.4):

$$E_{total} = \frac{1}{N} \sum_{i \in [n]} f_i \cdot |\hat{f}_i - f_i| = \frac{1}{N}(E_1 + E_2 + E_3)$$

For all groups of elements, let us decompose the expected weighted error $\mathbf{E_x}, \mathbf{x} \in \{\mathbf{1}, \mathbf{2}, \mathbf{3}\}$ they contribute to the final error E_{total} into two parts E_S and E_F. $\mathbf{E_S}$ describes the error obtained if we output the estimate in the sketch. $\mathbf{E_F}$ is the error incurred by outputting zero. Weighting each part, E_S and E_F, by the probability that these events occur (i.e., outputting zero vs. outputting the estimate from the big sketch table) leads to:

$$E_x = \sum_{i \in [G_x]} (p_S(G_x) \cdot E_S + p_F(G_x) \cdot E_F)$$

where $\mathbf{G_x}, \mathbf{x} \in \{\mathbf{1}, \mathbf{2}, \mathbf{3}\}$ is the group of elements under consideration, $\mathbf{p_S}(\mathbf{G_x})$ is the probability that the algorithm outputs the estimate from the big table for elements in group G_x, and $\mathbf{p_F}(\mathbf{G_x})$ is the probabilty that the algorithm outputs zero. Additionally, we know that for all groups:

$$E_S = \mathbb{E}[f_i \cdot |f_i - \tilde{f}_i|] = \frac{A}{i} \cdot \frac{A}{B \log B} = \frac{A^2}{iB \log B}$$

$$E_F = \mathbb{E}[f_i \cdot |f_i - 0|] = \frac{1}{i} \cdot \frac{A}{i} = \frac{A}{i^2}$$

Since for all $i \leq B \log B$ it holds that $E_S = \frac{A}{i} \cdot \frac{A}{B \log B} \leq \frac{A^2}{i^2} = E_F$, it is desirable to output the estimate from the big CountSketch table $\tilde{f}_i$ for these elements. For all other elements it follows that it is preferable to output zero.

Case 1: This case deals with all elements i, such that

$$i \leq \frac{A}{2\tau} = \frac{B}{2} \rightarrow f_i = \frac{A}{i} \geq 2\tau = \frac{2A}{B}$$

For these elements, it is desirable that the algorithm outputs the estimate from the big sketch table. Accordingly, the bad scenario is that the algorithm labels these frequent elements as rare and outputs zero, since

$$\forall i \leq \frac{A}{2\tau}: \qquad f_i = \frac{A}{i} \geq \frac{2A}{B} > \frac{A}{B \log B}$$

To upper bound the error E_1, we aim to minimize the upper bound of the probability that the algorithm outputs zero, p_F. Let $\hat{f}_i^{(k)}$ be the frequency estimate of the i^{th} element given by the k^{th} small CountSketch table. In the following, $\lesssim$ denotes inequality up to a constant factor.

The probability that a single estimate from a small table $\hat{f}_i^{(k)}$ is smaller than τ is:

$$\begin{aligned}
\Pr[\hat{f}_i^{(k)} \leq \tau] &= \Pr\left[-\hat{f}_i^{(k)} \geq -\tau\right] && \textit{Multiply both sides by -1} \\
&= \Pr\left[f_i - \hat{f}_i^{(k)} \geq f_i - \tau\right] && \textit{Add } f_i \textit{ to both sides} \\
&\leq \Pr\left[|f_i - \hat{f}_i^{(k)}| \geq f_i - \tau\right] && \textit{Take the absolute value of the left side} \\
&\leq \Pr\left[|f_i - \hat{f}_i^{(k)}| \geq 2\tau - \tau\right] && \textit{Lower bound } f_i \textit{ by } 2\tau\textit{, the minimum frequency in group 1} \\
&\leq \Pr\left[|f_i - \hat{f}_i^{(k)}| \geq \tau\right]
\end{aligned}$$

Given that term, we can now apply Theorem A.0.1:

$$\begin{aligned}
\Pr\left[|f_i - \hat{f}_i^{(k)}| \geq \tau\right] &\leq C'\left(\frac{\log(\tau\mathcal{B}')}{\tau\mathcal{B}'}\right)^2 && \textit{Apply Theorem A.0.1} \\
&\leq C'\left(\frac{\log(\frac{CA}{B}\cdot\frac{B\log B}{2T})}{\frac{CA}{B}\cdot\frac{B\log B}{2T}}\right)^2 && \textit{Plug in values: } \tau = \frac{A\cdot\mathcal{O}(1)}{B} \textit{ and } B' = \frac{B\log B}{2T} \\
&\leq C'\left(\frac{\log(\frac{CA}{B}\cdot\frac{B\log B}{2C''\log\log n})}{\frac{CA}{B}\cdot\frac{B\log B}{2C''\log\log n}}\right)^2 && \textit{Plug in value for } T, T = \mathcal{O}(\log\log n) \\
&\leq C'\left(\frac{2C''\log\left(\frac{AC}{2C''}\right)}{AC}\right)^2 && \textit{Reduce terms where possible} \\
&\leq 4C'C''^2\left(\frac{\log\left(\frac{AC}{2C''}\right)}{AC}\right)^2 && \textit{Move } 2C'' \textit{ out of parentheses} \\
&\lesssim \left(\frac{\log(AC)}{AC}\right)^2 \lesssim \frac{1}{8} && \textit{Upper bound by } \frac{1}{8} \\
&\lesssim e^{-\ln(8)} && \textit{Use trick that } e \textit{ and } \ln \textit{ equalize each other}
\end{aligned}$$

Let m be the number of small tables with estimates smaller than τ. If the algorithm outputs zero, at least half the estimates from these T small tables are smaller than the threshold τ, therefore, $m \geq \frac{T}{2}$. There are $\binom{T}{m}$ different combinations to choose m from the T estimates to be smaller than the threshold τ. Given that the estimates are independent, and the probability that a single estimate from a small table is smaller than τ is upper bounded by $e^{-\ln(8)}$ as

stated above, we get:

$$
\begin{aligned}
p_F(G_1) &= \sum_{m=T/2}^{T} \binom{T}{m} \cdot Pr[\hat{f}_i^{(k)} \leq \tau]^m \cdot Pr[\hat{f}_i^{(k)} > \tau]^{T-m} \\
&\leq \sum_{m=T/2}^{T} \binom{T}{m} \cdot Pr[\hat{f}_i^{(k)} \leq \tau]^m && \textit{Omit } Pr[\hat{f}_i^{(k)} > \tau]^{T-m} \\
&\leq \frac{T}{2} \cdot \binom{T}{T/2} \cdot Pr[\hat{f}_i^{(k)} \leq \tau]^{T/2} && \textit{Upper bound the single addends } \left(\textit{max. for } m = \frac{T}{2}\right) \\
&\leq \frac{T}{2} \cdot \left(\frac{2eT}{T}\right)^{T/2} \cdot Pr[\hat{f}_i^{(k)} \leq \tau]^{T/2} && \textit{Upper bound binom. coefficient by } \binom{n}{k} \leq \left(\frac{en}{k}\right)^k \\
&\leq \frac{T}{2} \cdot (2e)^{T/2} \cdot Pr[\hat{f}_i^{(k)} \leq \tau]^{T/2} && \textit{Reduce terms} \\
&\lesssim \frac{T}{2} \cdot (2e)^{T/2} \cdot e^{-\ln(8)\frac{T}{2}} && \textit{Plug in upper bound for } Pr[\hat{f}_i^{(k)} \leq \tau] \\
&\lesssim \frac{T}{2} \cdot (2e)^{T/2} \cdot e^{-\Theta(T)} && \textit{Rewrite } e^{-\ln(8)\frac{T}{2}} \textit{ as } e^{-\Theta(T)} \\
&\lesssim e^{-\Theta(T)} && \textit{Omit further constant parts of term} \\
&\lesssim e^{-\Theta(\log\log n)} && \textit{Plug in value for } T \\
&\lesssim \frac{1}{\log^C(n)} && \textit{Rewrite term; plug in constant } C \textit{ for } \Theta \\
&\lesssim \frac{1}{\log^{100}(n)} && \textit{Plug in concrete value for } C
\end{aligned}
$$

Plugging in the obtained upper bound for $p_F(G_1)$, the probability that the bad scenario occurs, i.e., the algorithm outputs zero, and upper-bounding the probability for the good scenario

$p_F(G_1)$ by one, we get:

$$
\begin{aligned}
E_1 &= \sum_{i=1}^{\frac{A}{2\tau}} \mathbb{E}[f_i \cdot |f_i - \hat{f}_i|] \\
&= \sum_{i=1}^{\frac{A}{2\tau}} \left(p_F(G_1) \cdot E_F + p_S(G_1) \cdot E_S\right) \\
&\lesssim \sum_{i=1}^{\frac{A}{2\tau}} \left(p_F(G_1) \cdot \frac{A^2}{i^2} + p_S(G_1) \cdot \frac{A^2}{iB\log B}\right) && \textit{Plug in } E_S = \frac{A^2}{iB\log B} \textit{ and } E_F = \frac{A^2}{i^2} \\
&\lesssim \sum_{i=1}^{\frac{A}{2\tau}} \left(\frac{1}{\log^{100}(n)} \cdot \frac{A^2}{i^2} + \frac{A^2}{iB\log B}\right) && \textit{Plug in obtained upper bound for } p_F(G_1) \\
& && \textit{and upper bound } p_S(G_1) \textit{ by 1} \\
&\lesssim \frac{A^2}{\log^{100}(n)} \sum_{i=1}^{\frac{A}{2\tau}} \frac{1}{i^2} + \frac{A^2}{B\log B} \sum_{i=1}^{\frac{A}{2\tau}} \frac{1}{i} && \textit{Move constants out of sum} \\
&\lesssim \frac{A^2}{\log^{100}(n)} \int_1^{\frac{A}{2\tau}} \frac{1}{x^2}\, dx + \frac{A^2}{B\log B} \cdot \log\left(\frac{A}{2\tau}\right) && \textit{Upper bound left sum by integral and right sum} \\
& && \textit{by harmonic series } H(n) = \sum_{x=1}^{n} \frac{1}{x} \lesssim \log(n) \\
&\lesssim \frac{A^2}{\log^{100}(n)} \cdot \left(1 - \frac{2\tau}{A}\right) + \frac{A^2}{B\log B} \cdot \log\left(\frac{A}{2\tau}\right) && \textit{Calculate integral} \\
&\lesssim \frac{A^2}{\log^{100}(n)} \cdot \left(1 - \frac{2}{B}\right) + \frac{A^2}{B\log B} \cdot \log\left(\frac{B}{2}\right) && \textit{Plug in value for } \tau \\
&\lesssim A^2 \left(\frac{B-2}{\log^{100}(n)B} + \frac{\log(B/2)}{B\log B}\right) && \textit{Pull out } A^2 \textit{ and pull apart logarithmic term} \\
&= \mathcal{O}\left(\frac{A^2\log(B/2)}{B\log B}\right) = \mathcal{O}\left(\frac{A^2}{B}\right) && \textit{Only keep dominating part and reduce}
\end{aligned}
$$

Case 2: This case deals with all elements i, such that

$$i \geq \frac{2A}{\tau} = 2B \rightarrow f_i = \frac{A}{i} \leq \frac{\tau}{2} = \frac{A}{2B}$$

Note that not for all elements in this group, the best and worst cases are the same: For some elements i such that $2B \leq i \leq B\log B$, it is indeed still preferable to output the estimate from the big sketch table. However, for most elements in group two, the best and worst case scenarios are just inverse to the first case: It is desirable that the algorithm outputs zero, while the bad scenario is that the algorithm outputs the estimate from the big sketch table.

Following the same reasoning as in the first case, we can upper bound the error E_2 by upper bounding the probability that the algorithm outputs the estimate from the big sketch table, p_S. To do so, let us first consider the probability that a single estimate from the small tables is larger than τ:

$$\begin{aligned}
\Pr[\hat{f}_i^{(k)} \geq \tau] &= \Pr\left[\hat{f}_i^{(k)} - f_i \geq \tau - f_i\right] && \textit{Subtract } f_i \textit{ from both sides} \\
&\leq \Pr\left[|\hat{f}_i^{(k)} - f_i| \geq \tau - f_i\right] && \textit{Take the absolute value of the left side} \\
&\leq \Pr\left[|\hat{f}_i^{(k)} - f_i| \geq \tau - \frac{\tau}{2}\right] && \textit{Upper bound } f_i \textit{ by } \frac{\tau}{2}\textit{, the max. frequency in group 2} \\
&\leq \Pr\left[|\hat{f}_i^{(k)} - f_i| \geq \frac{\tau}{2}\right]
\end{aligned}$$

Like in the first case, we can now apply Theorem A.0.1:

$$\begin{aligned}
\Pr\left[|f_i - \hat{f}_i^{(k)}| \geq \frac{\tau}{2}\right] &\leq C'\left(\frac{\log(\frac{\tau}{2}B')}{\frac{\tau}{2}B'}\right)^2 && \textit{Apply Theorem A.0.1} \\
&\leq C'\left(\frac{\log(\frac{CA}{2B}\cdot\frac{B\log B}{2T})}{\frac{CA}{2B}\cdot\frac{B\log B}{2T}}\right)^2 && \textit{Plug in values: } \tau = \frac{A\mathcal{O}(1)}{B} \textit{ and } B' = \frac{B\log B}{2T} \\
&\leq C'\left(\frac{\log(\frac{CA}{2B}\cdot\frac{B\log B}{2C''\log\log n})}{\frac{CA}{2B}\cdot\frac{B\log B}{2C''\log\log n}}\right)^2 && \textit{Plug in value for } T, T = \mathcal{O}(\log\log n) \\
&\leq C'\left(\frac{4C''\log\left(\frac{AC}{4C''}\right)}{AC}\right)^2 && \textit{Reduce terms where possible} \\
&\leq 16C'C''^2\left(\frac{\log\left(\frac{AC}{4C''}\right)}{AC}\right)^2 && \textit{Move constants out of parentheses} \\
&\lesssim \left(\frac{\log(AC)}{AC}\right)^2 \lesssim \frac{1}{8} && \textit{Omit irrelevant constants and upper bound by } \frac{1}{8} \\
&\lesssim e^{-\ln(8)} && \textit{Use trick that } e \textit{ and } \ln \textit{ equalize each other}
\end{aligned}$$

We can now determine the probability that the algorithm outputs the estimate from the big

table, $p_S(G_2)$. The steps follow the same logic as in case one:

$$
\begin{aligned}
p_S(G_2) &= \sum_{m=T/2}^{T} \binom{T}{m} \cdot Pr[\hat{f}_i^{(k)} \geq \tau]^m \cdot Pr[\hat{f}_i^{(k)} < \tau]^{T-m} \\
&\leq \sum_{m=T/2}^{T} \binom{T}{m} \cdot Pr[\hat{f}_i^{(k)} \geq \tau]^m && \textit{Omit } Pr[\hat{f}_i^{(k)} < \tau]^{T-m} \\
&\leq \frac{T}{2} \cdot \binom{T}{T/2} \cdot Pr[\hat{f}_i^{(k)} \geq \tau]^{T/2} && \textit{Upper bound the single addends } \left(\textit{max. for } m = \frac{T}{2}\right) \\
&\leq \frac{T}{2} \cdot \left(\frac{2eT}{T}\right)^{T/2} \cdot Pr[\hat{f}_i^{(k)} \geq \tau]^{T/2} && \textit{Upper bound binom. coefficient by } \binom{n}{k} \leq \left(\frac{en}{k}\right)^k \\
&\leq \frac{T}{2} \cdot (2e)^{T/2} \cdot Pr[\hat{f}_i^{(k)} \geq \tau]^{T/2} && \textit{Reduce terms} \\
&\lesssim \frac{T}{2} \cdot (2e)^{T/2} \cdot e^{-\ln(8)\frac{T}{2}} && \textit{Plug in upper bound for } Pr[\hat{f}_i^{(k)} \geq \tau] \\
&\lesssim \frac{T}{2} \cdot (2e)^{T/2} \cdot e^{-\Theta(T)} && \textit{Rewrite } e^{-\ln(8)\frac{T}{2}} \textit{ as } e^{-\Theta(T)} \\
&\lesssim e^{-\Theta(T)} && \textit{Omit further constant parts of term} \\
&\lesssim e^{-\Theta(\log\log n)} && \textit{Plug in value for } T \\
&\lesssim \frac{1}{\log^C(n)} && \textit{Rewrite term; plug in constant } C \textit{ for } \Theta \\
&\lesssim \frac{1}{\log^{100}(n)} && \textit{Plug in concrete value for } C
\end{aligned}
$$

Hence:

$$
\begin{aligned}
E_2 &= \sum_{i=\frac{2A}{\tau}}^{n} \mathbb{E}[f_i \cdot |f_i - \hat{f}_i|] \\
&\lesssim \sum_{i=\frac{2A}{\tau}}^{n} (p_S(G_2) \cdot E_S + p_F(G_2) \cdot E_F) \\
&\lesssim \sum_{i=\frac{2A}{\tau}}^{n} \left(p_S(G_2) \cdot \frac{A^2}{iB} + p_F(G_2) \cdot \frac{A^2}{i^2} \right) && \textit{Plug in } E_S = \frac{A^2}{iB \log B} \textit{ and } E_F = \frac{A^2}{i^2} \\
&\lesssim \sum_{i=\frac{2A}{\tau}}^{n} \left(\frac{A^2}{\log(n)^{100}} \cdot \frac{1}{iB \log B} + \frac{A^2}{i^2} \right) && \textit{Plug in upper bound for } p_F(G_2) \\
& && \textit{and upper bound } p_S(G_2) \textit{ by 1} \\
&\lesssim \frac{A^2}{B \log B \cdot \log^{100}(n)} \cdot \sum_{i=\frac{2A}{\tau}}^{n} \frac{1}{i} + A^2 \cdot \sum_{i=\frac{2A}{\tau}}^{n} \frac{1}{i^2} && \textit{Split sum and pull out constant parts} \\
&\lesssim \frac{A^2}{B \log B \cdot \log^{100}(n)} \cdot \log(n) + A^2 \cdot \int_{\frac{2A}{\tau}}^{n} \frac{1}{x^2}\, dx && \textit{Upper bound right sum by integral and left side by} \\
& && \textit{harmonic series } H(n) = \sum_{x=1}^{n} \frac{1}{x} \lesssim \log(n) \\
&\lesssim A^2 \cdot \left[\frac{1}{B \log B \cdot \log^{99}(n)} + \left(-\frac{1}{n} + \frac{\tau}{2A} \right) \right] && \textit{Pull out } A^2 \textit{ and calculate integral} \\
&\lesssim A^2 \cdot \left[\frac{1}{B \log B \cdot \log^{99}(n)} + \frac{1}{2B} \right] && \textit{Plug in value for } \tau \textit{ and reduce} \\
&= \mathcal{O}\left(\frac{A^2}{B} \right) && \textit{Only keep dominating part}
\end{aligned}
$$

Case 3: The last case to consider is the case of all elements i such that

$$\frac{A}{2\tau} \leq i \leq \frac{2A}{\tau} \rightarrow 2\tau \geq f_i \geq \frac{\tau}{2}$$

In the worst case, the algorithm outputs zero for all these elements, since

$$\frac{A}{i} \geq \frac{\tau}{2} \gtrsim \frac{A}{2B} > \frac{A}{B \log B}$$

Therefore, we can upper bound the error by assuming the worst-case scenario for all elements:

$$
\begin{aligned}
E_3 &= \mathbb{E}[f_i \cdot |f_i - \hat{f}_i|] \\
&\lesssim \sum_{i=\frac{A}{2\tau}}^{\frac{2A}{\tau}} (p_S(G_3) \cdot E_S + p_F(G_3) \cdot E_F) \\
&\lesssim \sum_{i=\frac{A}{2\tau}}^{\frac{2A}{\tau}} \frac{A^2}{i^2} && \textit{Assume worst case, i.e. } p_F(G_3) = 1 \\
&\lesssim A^2 \cdot \int_{\frac{A}{2\tau}}^{\frac{2A}{\tau}} \frac{1}{x^2}\, dx && \textit{Upper bound sum by integral} \\
&\lesssim A^2 \left(-\frac{\tau}{2} + 2\tau\right) && \textit{Calculate the integral} \\
&\lesssim A^2 \left(-\frac{1}{2B} + \frac{2}{B}\right) && \textit{Plug in value for } \tau \\
&= \mathcal{O}\left(\frac{A^2}{B}\right)
\end{aligned}
$$

Finally, we can combine the three cases to obtain the total expected weighted error:

$$
\begin{aligned}
E_{total} &= \frac{1}{N}(E_1 + E_2 + E_3) = \frac{1}{N}\left[\mathcal{O}\left(\frac{A^2}{B}\right) + \mathcal{O}\left(\frac{A^2}{B}\right) + \mathcal{O}\left(\frac{A^2}{B}\right)\right] \\
&= \frac{1}{N} \cdot \mathcal{O}\left(\frac{A^2}{B}\right) = \frac{1}{N} \cdot \mathcal{O}\left(\frac{N^2}{\log^2(n)}\frac{1}{B}\right) \\
&= \mathcal{O}\left(\frac{N}{\log^2(n)}\frac{1}{B}\right)
\end{aligned}
$$

□

This concludes the proof that, indeed, a standard CountSketch++ using a space budget of $B' = \mathcal{O}(B \log B)$ achieves the same weighted error as its learned counterpart with space budget B.

B. Additional Results

B.1. Experiment 1.1

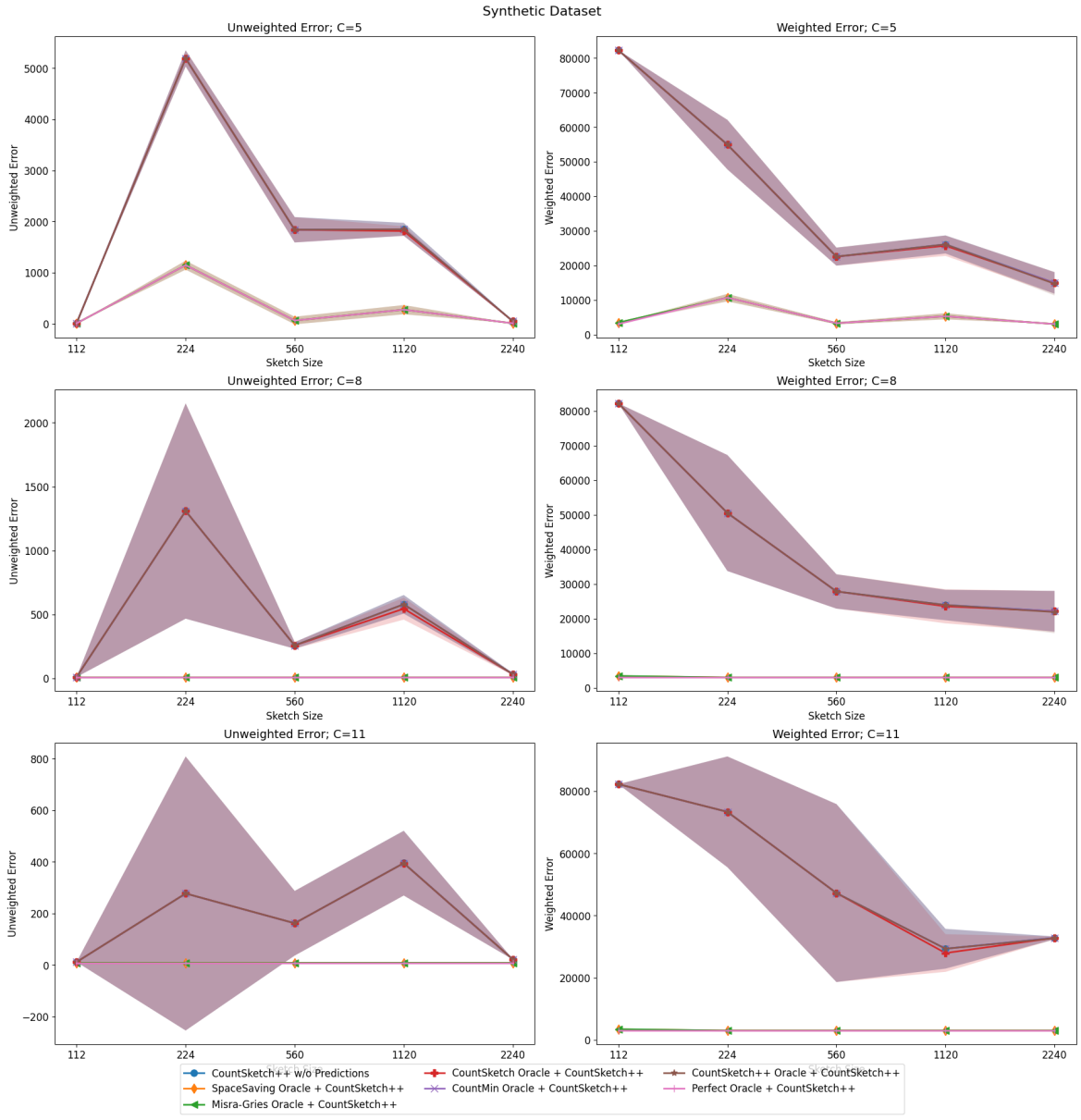


Figure B.1.: Complete results for experiment 1.1 on synthetic data (Oracle + learned CountSketch++ (2 Passes))

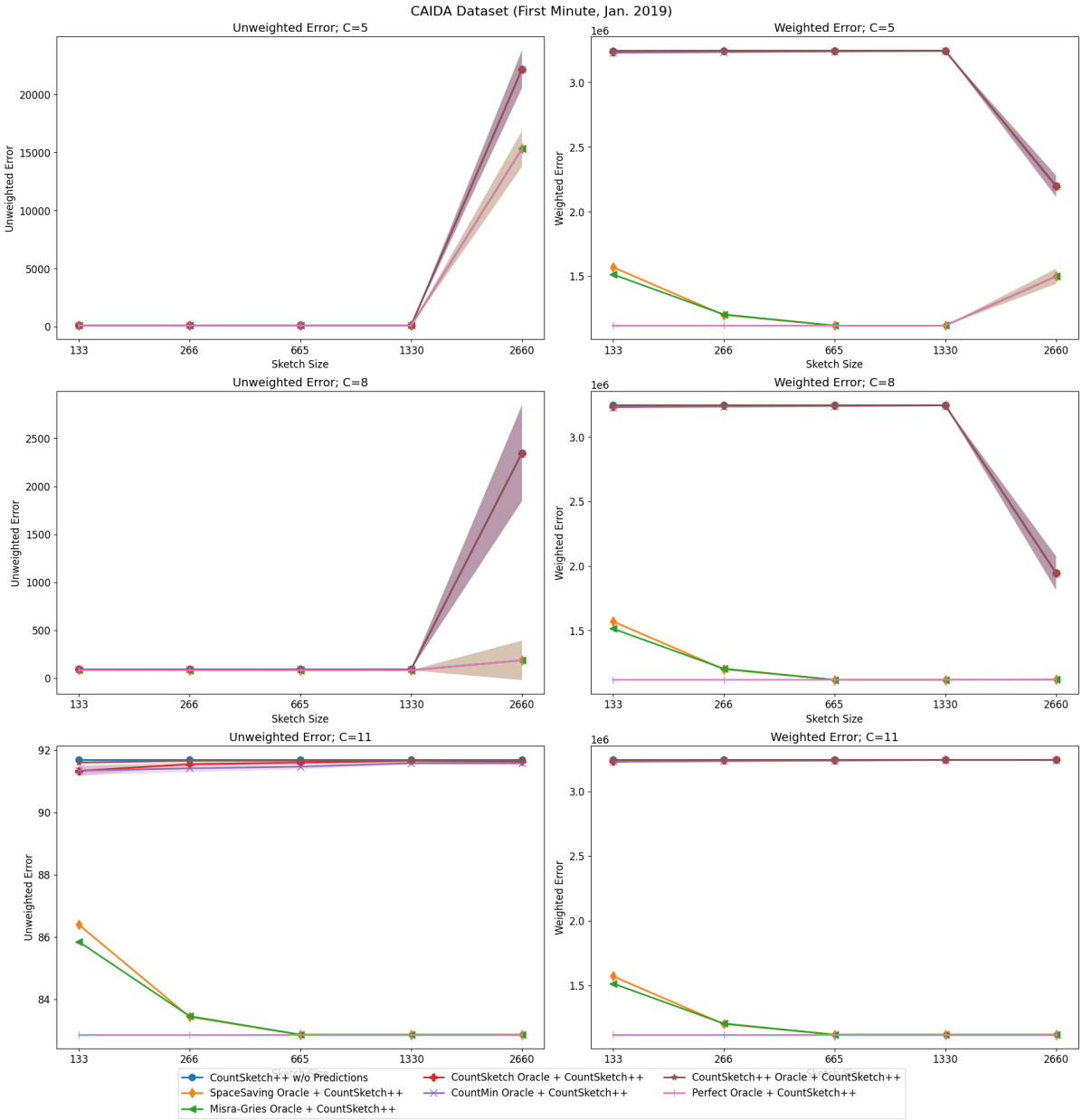


Figure B.2.: Complete results for experiment 1.1 on CAIDA data (Oracle + learned CountSketch++ (2 Passes))

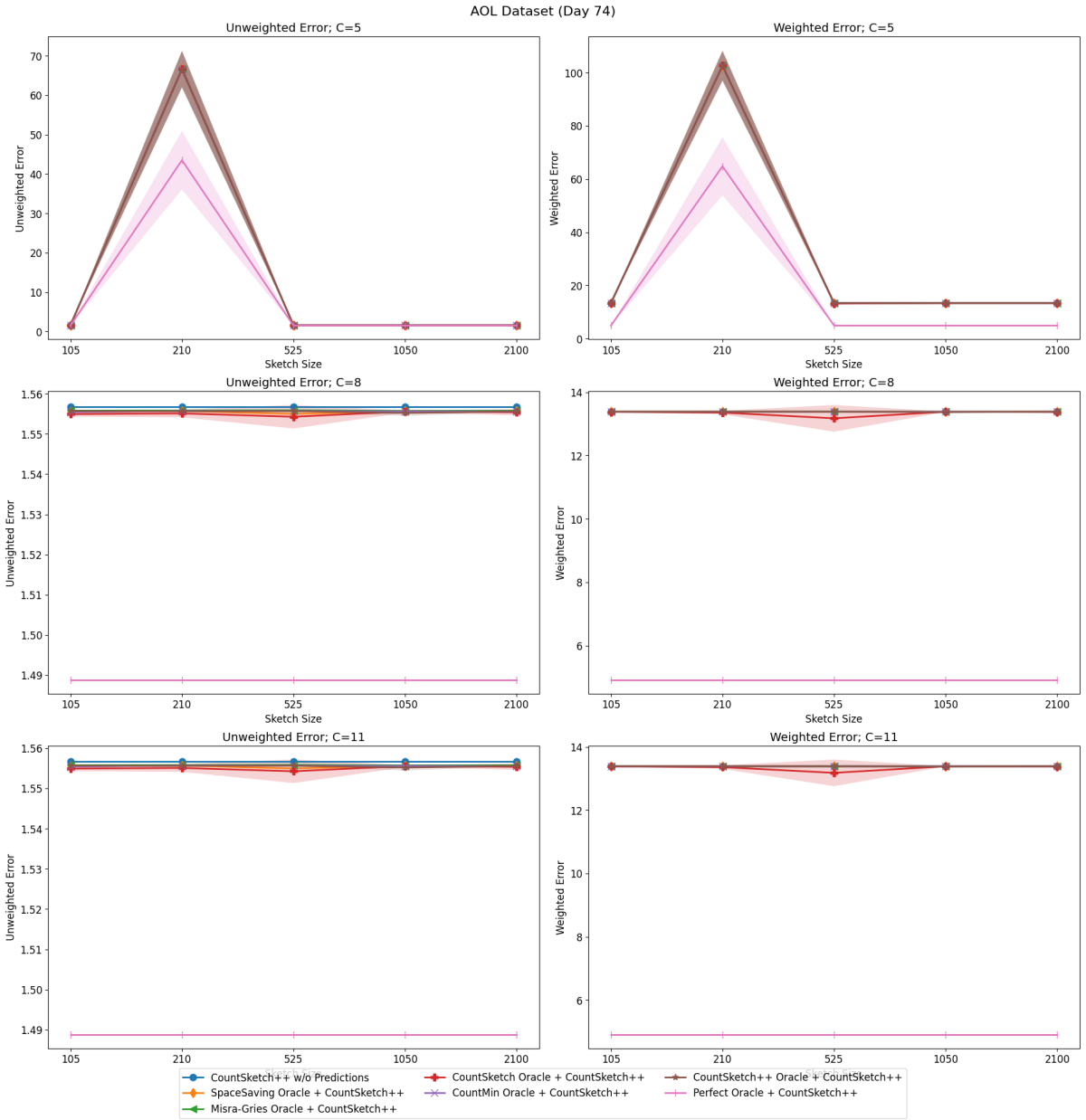


Figure B.3.: Complete results for experiment 1.1 on AOL data (Oracle + learned CountSketch++ (2 Passes))

B.2. Experiment 1.2

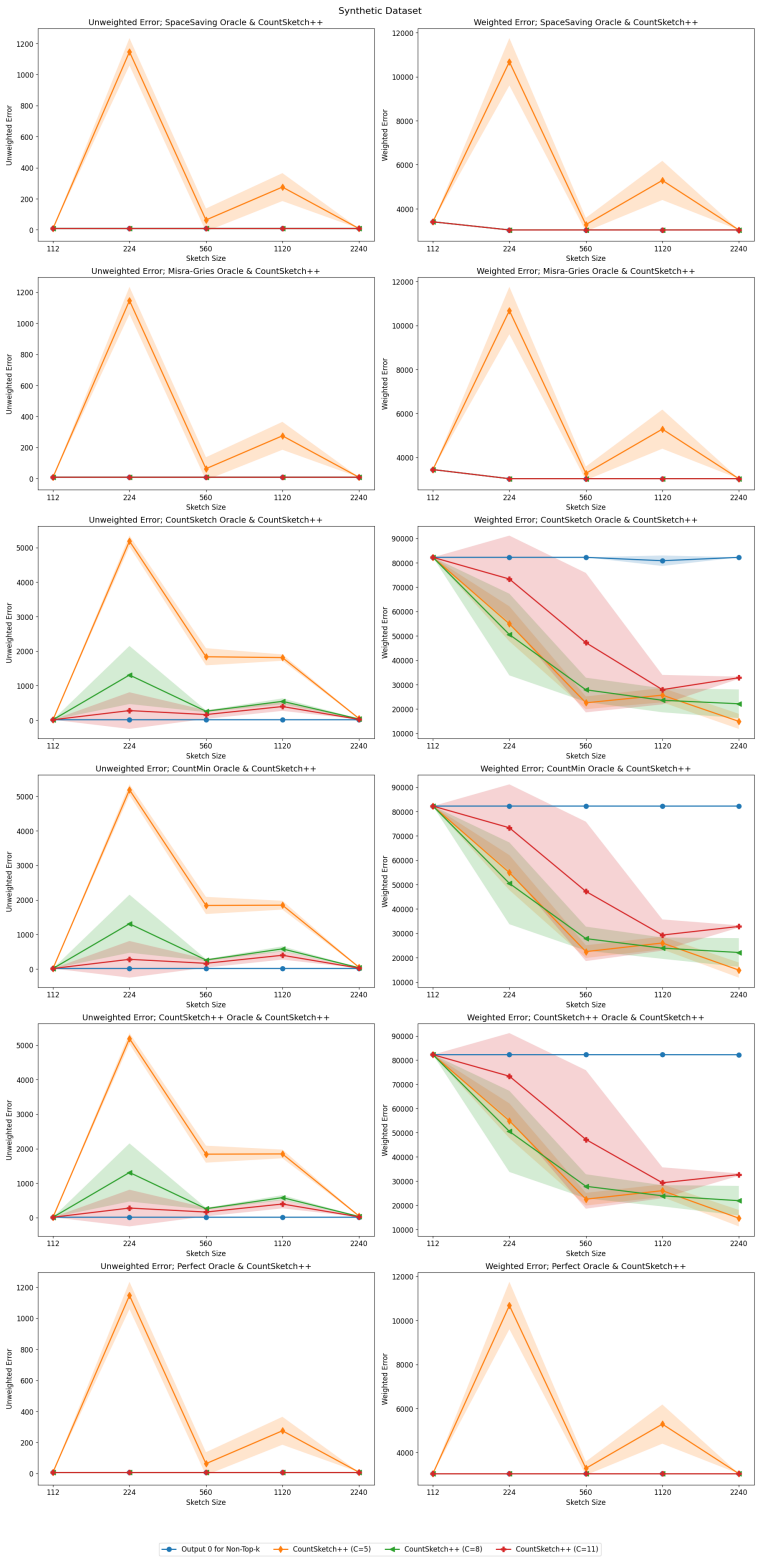


Figure B.4.: Complete results for experiment 1.2 on synthetic data (Oracle + CountSketch++ vs. Oracle + Trivial Mechanism (2 Passes))

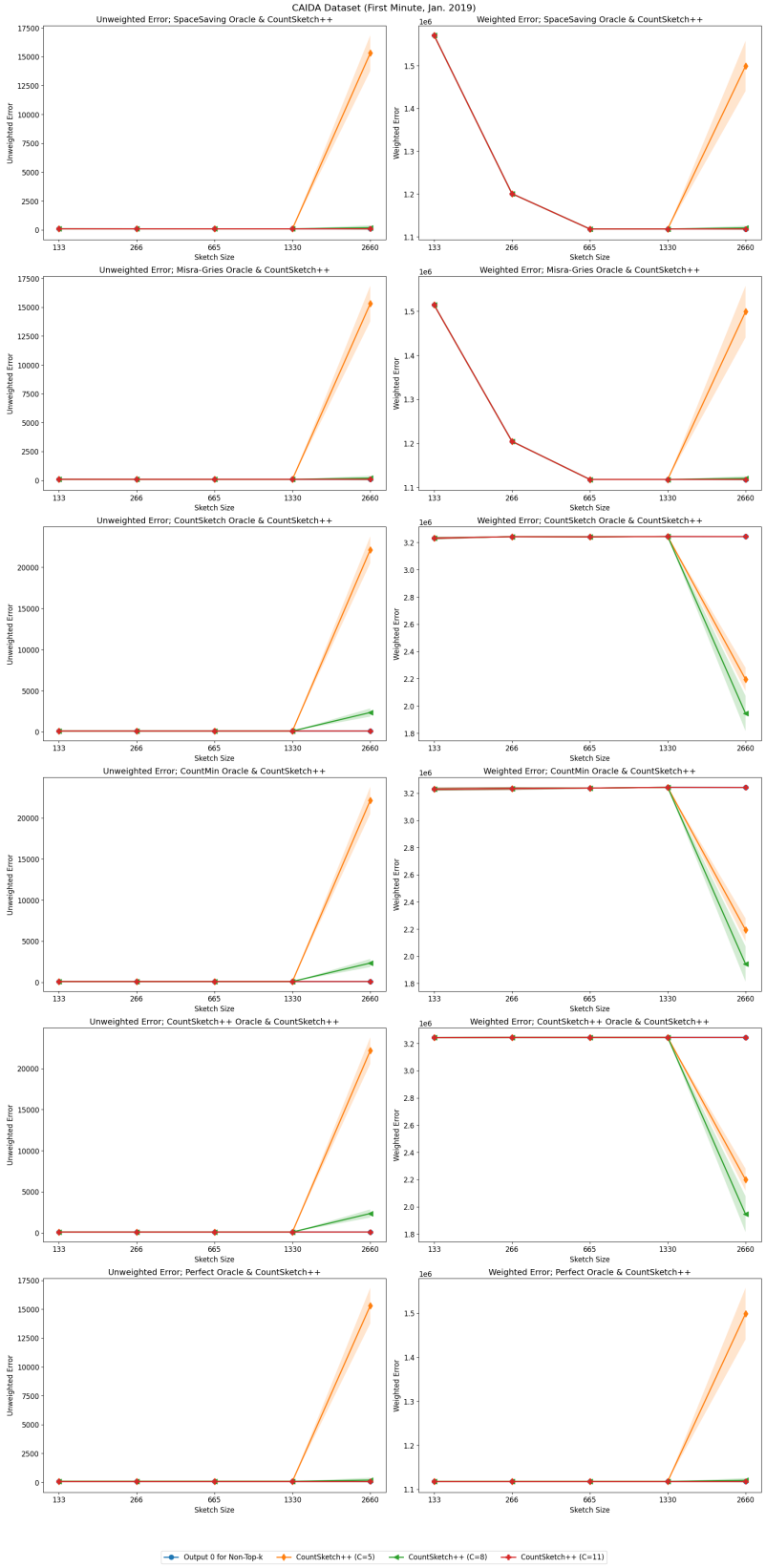


Figure B.5.: Complete results for experiment 1.2 on CAIDA data (Oracle + CountSketch++ vs. Oracle + Trivial Mechanism (2 Passes))

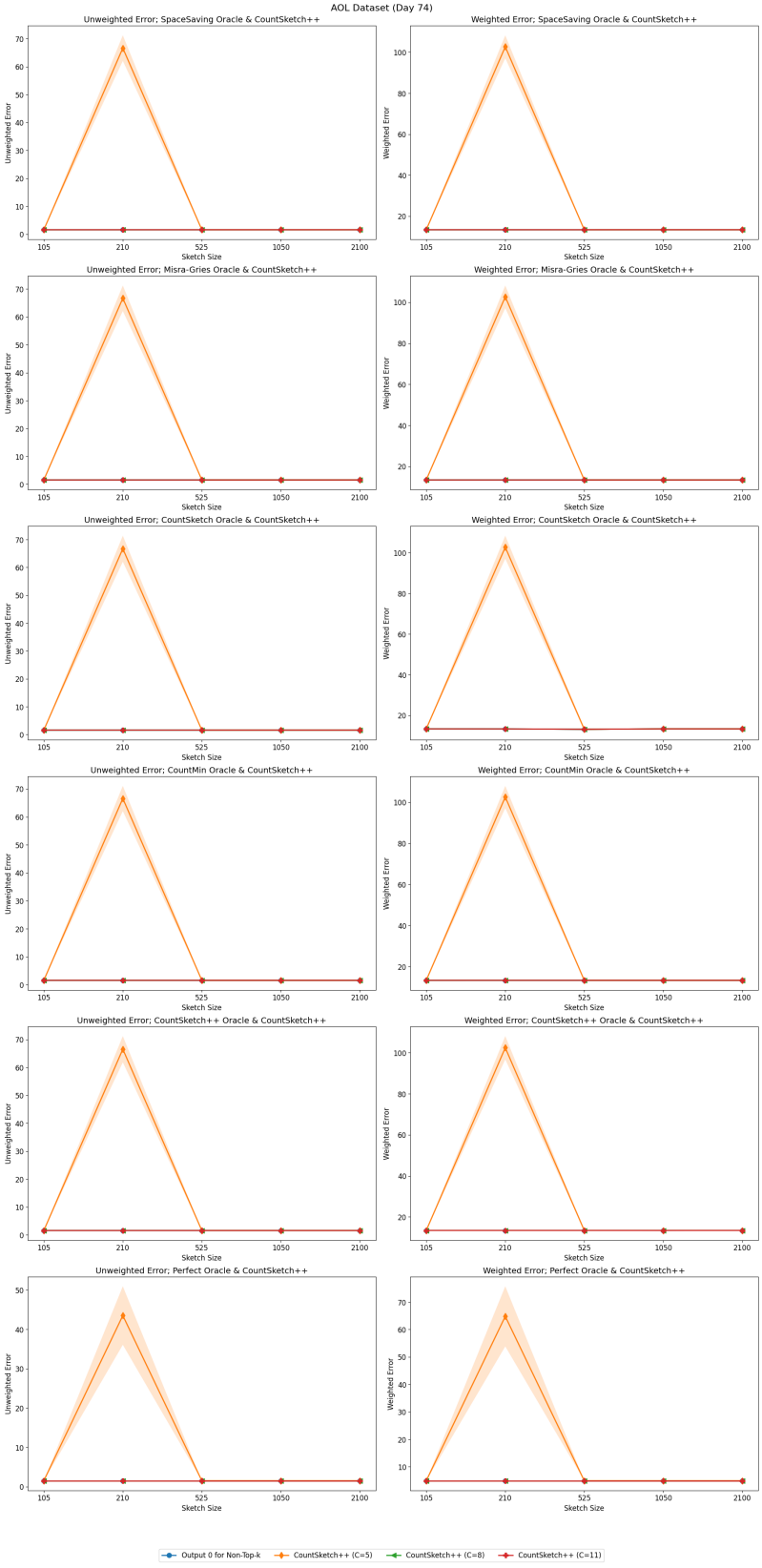


Figure B.6.: Complete results for experiment 1.2 on AOL data (Oracle + CountSketch++ vs. Oracle + Trivial Mechanism (2 Passes))

B.3. Experiment 2.1

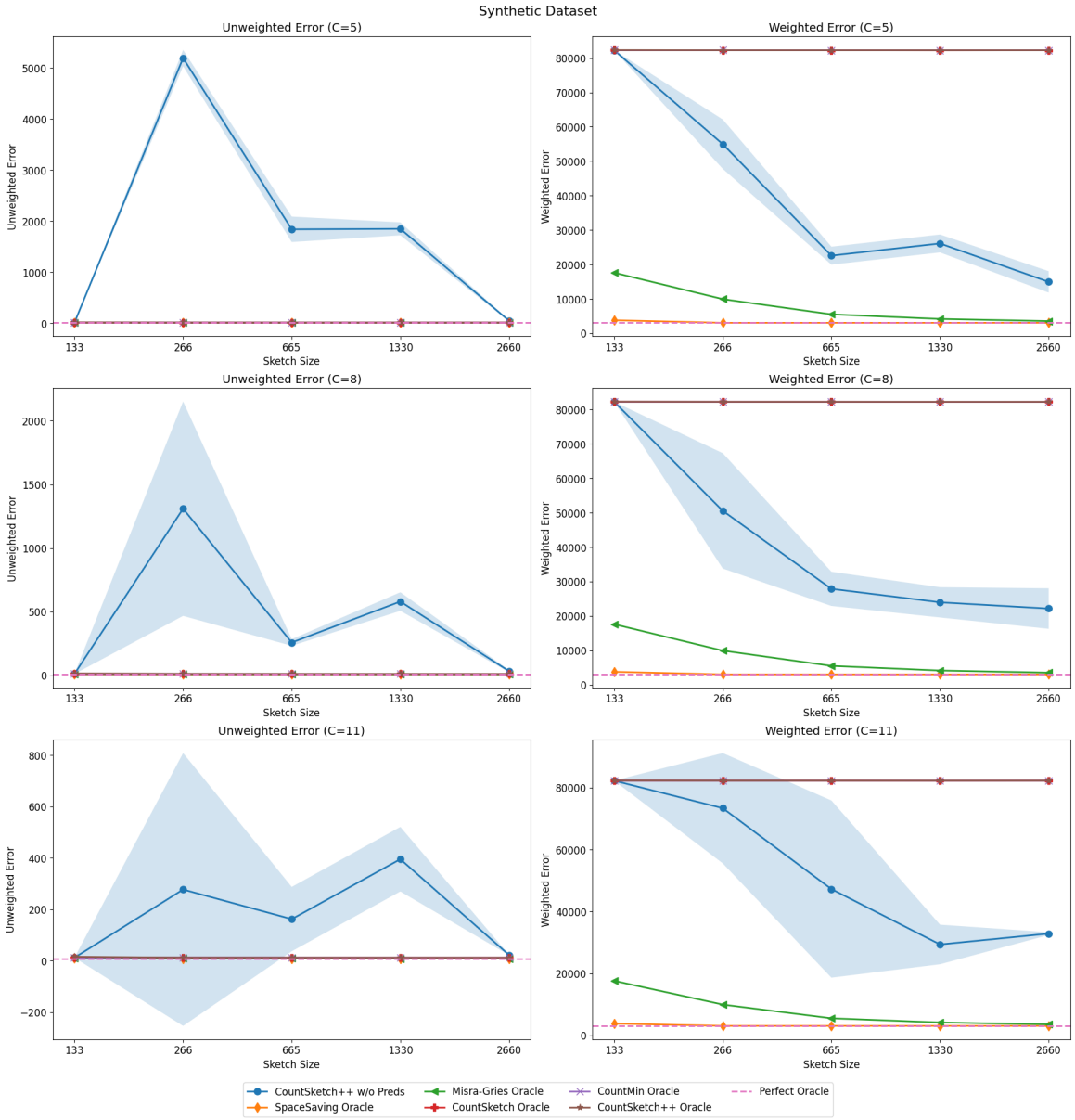


Figure B.7.: Complete results for experiment 2.1 on synthetic data (Oracle counts only (1 Passes))

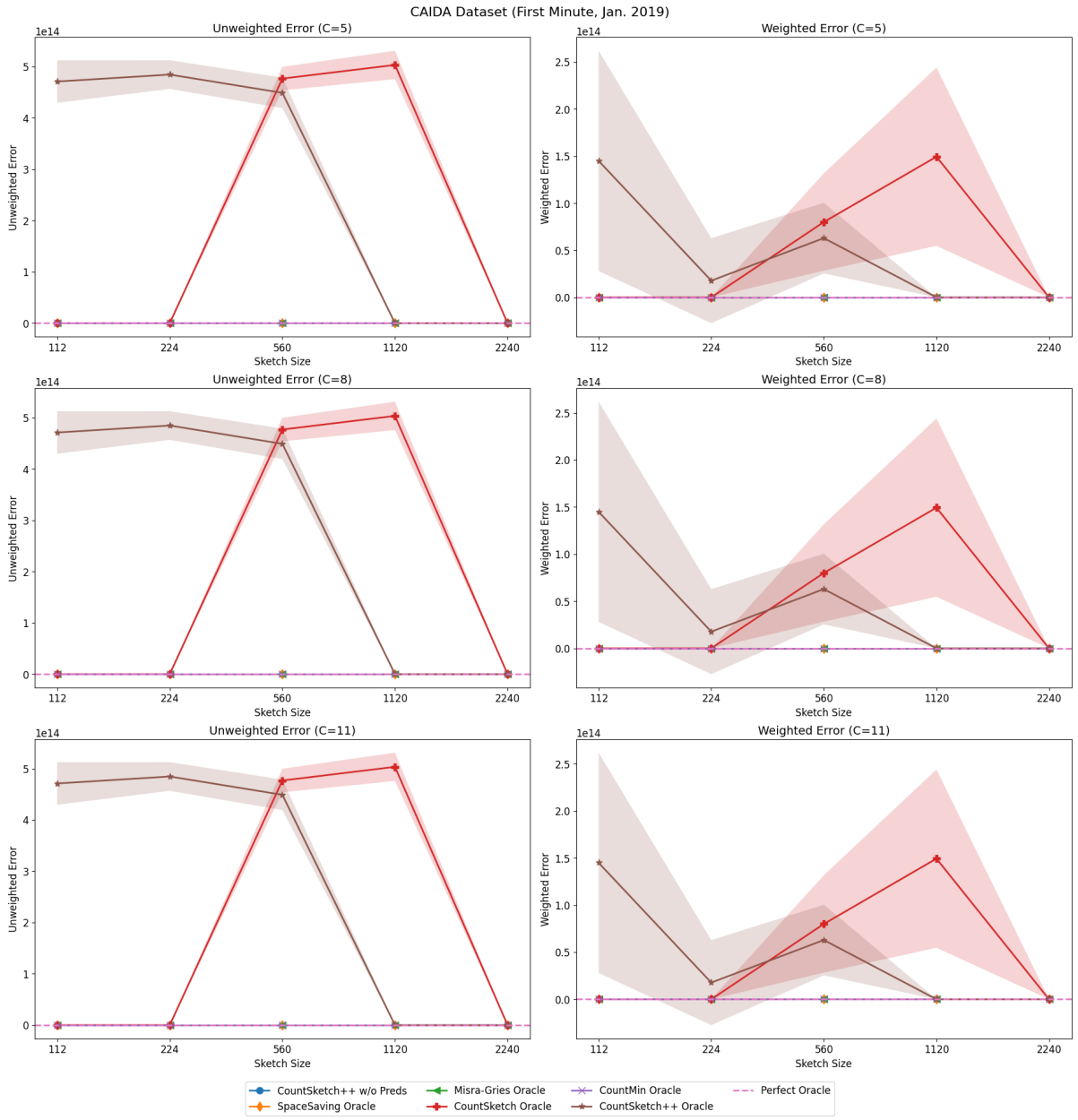


Figure B.8.: Complete results for experiment 2.1 on CAIDA data (Oracle counts only (1 Passes))

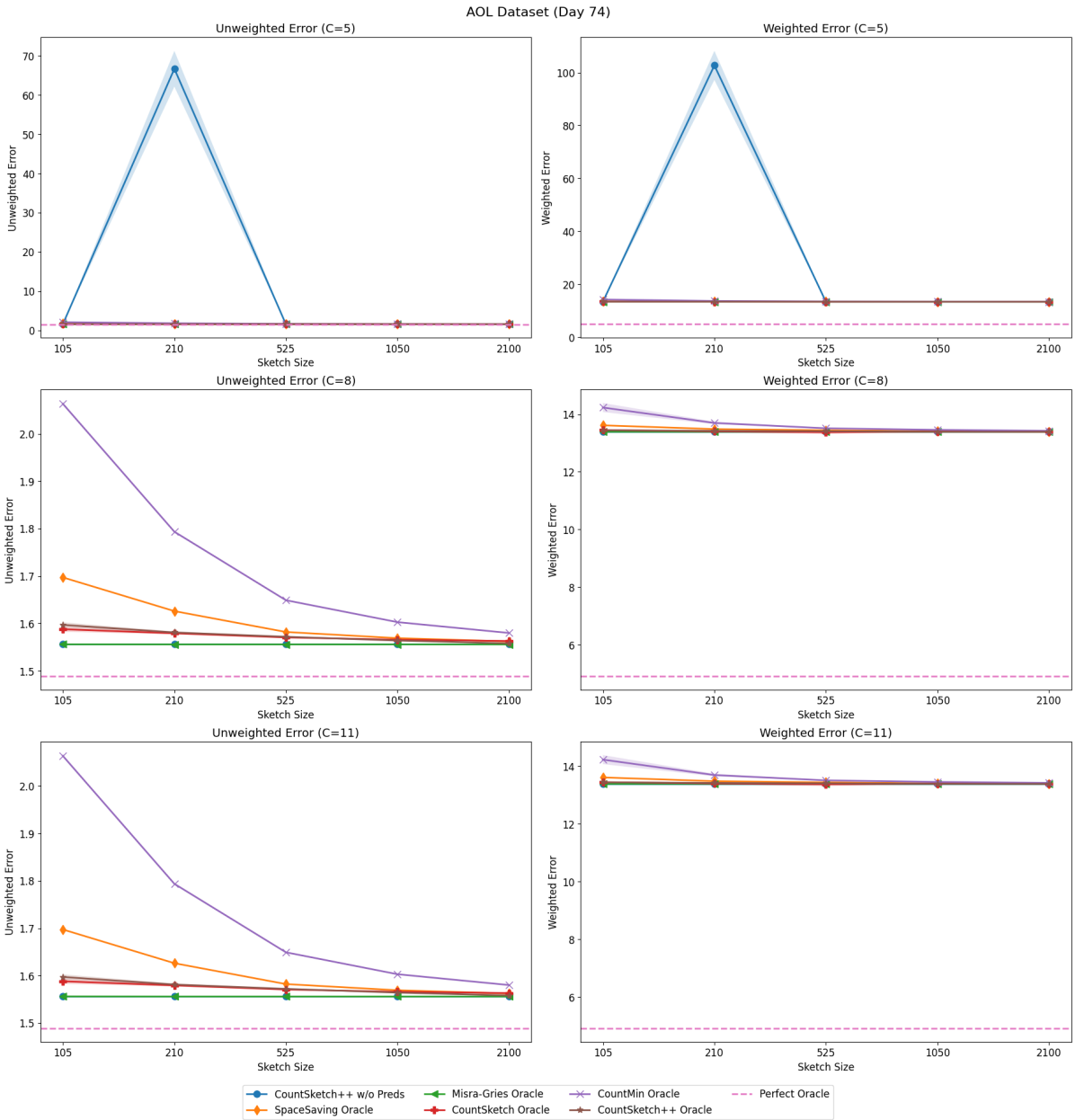


Figure B.9.: Complete results for experiment 2.1 on AOL data (Oracle counts only (1 Passes))

B.4. Experiment 2.2

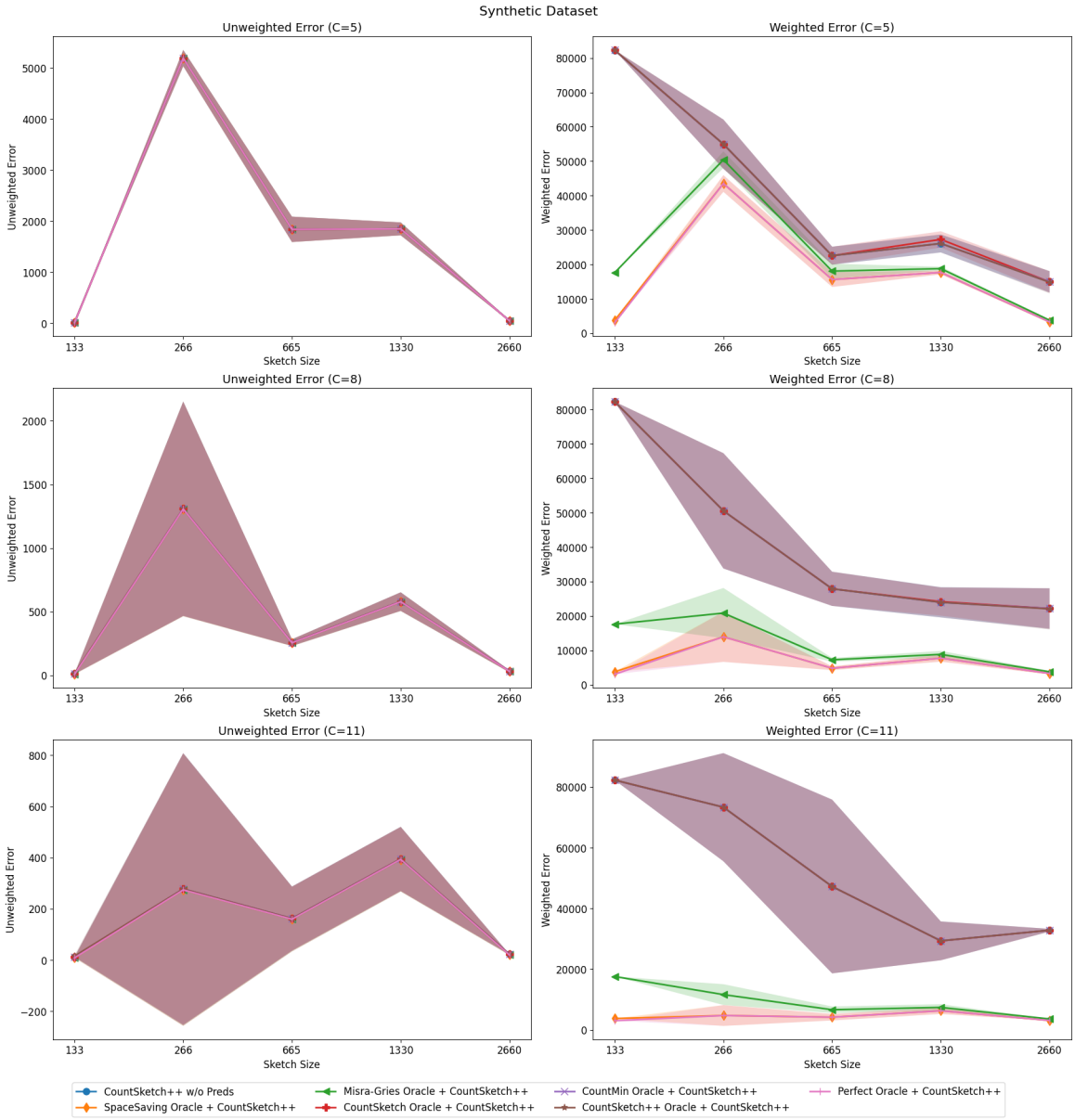


Figure B.10.: Complete results for experiment 2.2 on synthetic data (Augmented Oracles: Oracle + CountSketch++ (1 Passes))

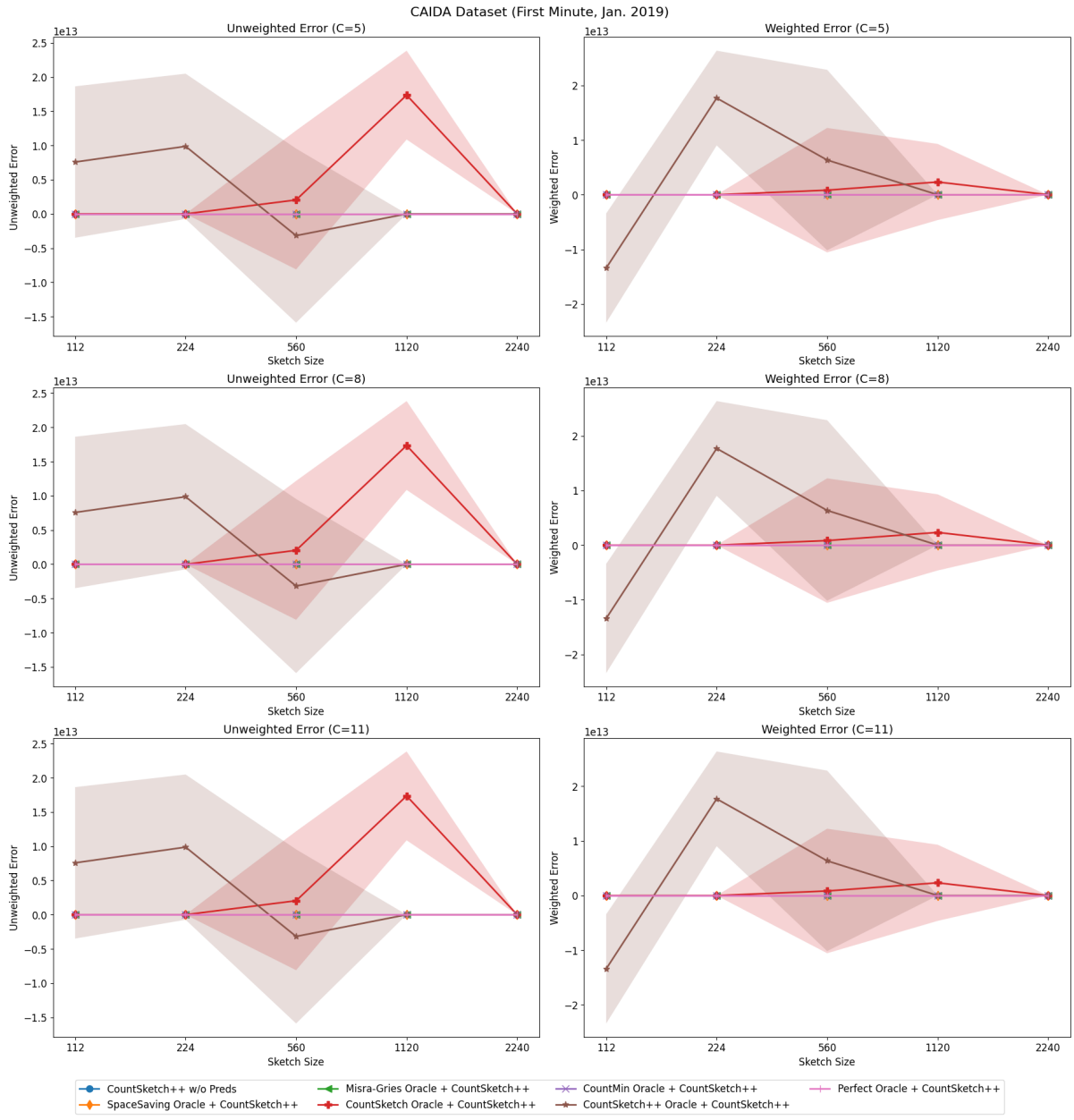


Figure B.11.: Complete results for experiment 2.2 on CAIDA data (Augmented Oracles: Oracle + CountSketch++ (1 Passes))

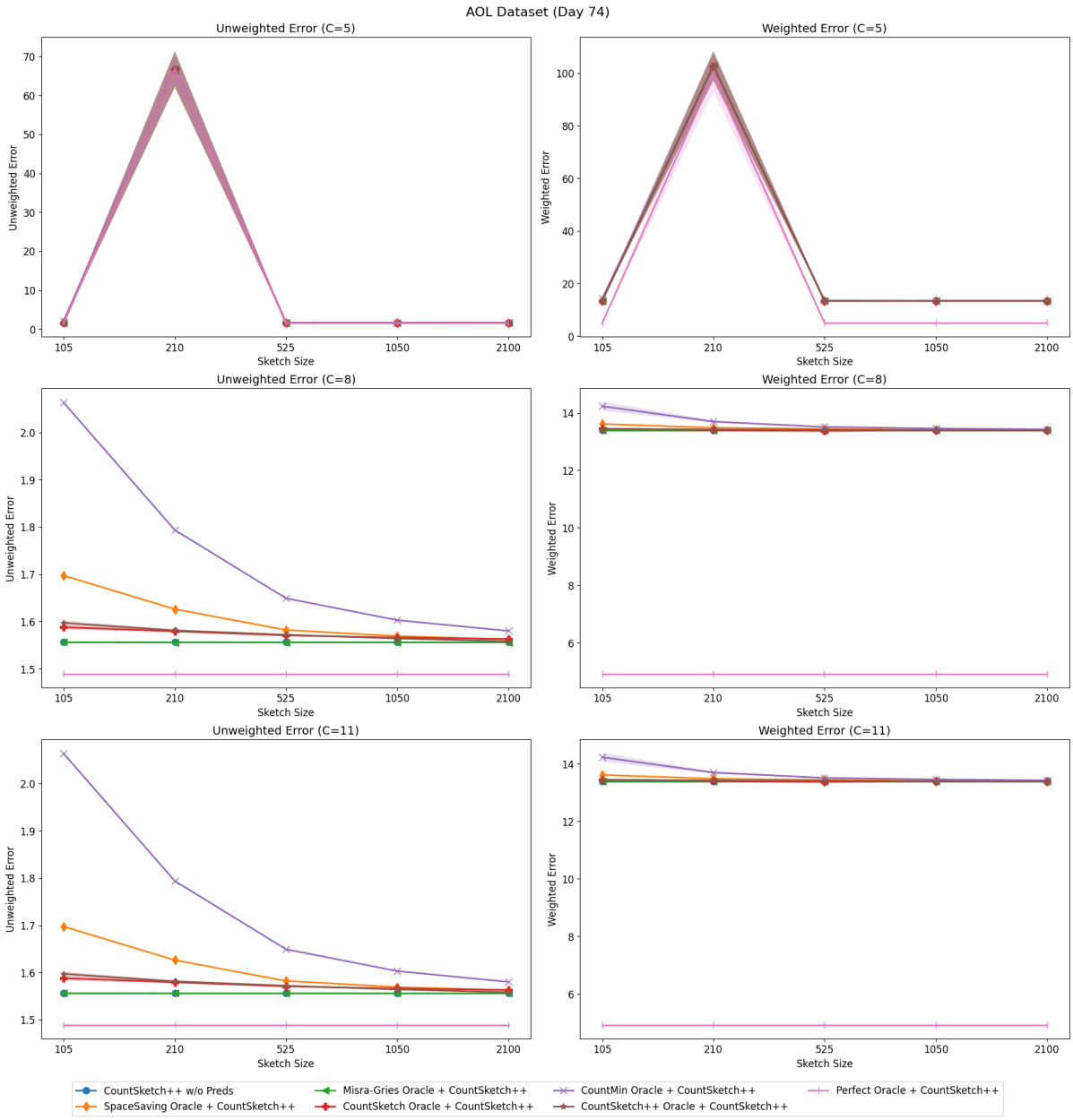


Figure B.12.: Complete results for experiment 2.2 on AOL data (Augmented Oracles: Oracle + CountSketch++ (1 Passes))

List of Figures

List of Tables

Bibliography

[Aam+23] A. Aamand, J. Y. Chen, H. L. Nguyễn, S. Silwal, and A. Vakilian. "Improved frequency estimation algorithms with and without predictions." In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. NIPS '23. New Orleans, LA, USA: Curran Associates Inc., 2023.

[Aam+25] A. Aamand, J. Y. Chen, S. Gollapudi, S. Silwal, and H. WU. "Learning-Augmented Frequent Directions." In: *The Thirteenth International Conference on Learning Representations*. 2025.

[Afe+16] Y. Afek, A. Bremler-barr, E. Cohen, S. Landau Feibish, and M. Shagam. "Efficient Distinct Heavy Hitters for DNS DDoS Attack Detection." In: (Dec. 2016). DOI: 10.48550/arXiv.1612.02636.

[Ber+10] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss. "Space-optimal heavy hitters with strong error bounds." In: *ACM Trans. Database Syst.* 35.4 (Oct. 2010). ISSN: 0362-5915. DOI: 10.1145/1862919.1862923.

[CAI19] CAIDA. *The CAIDA UCSD Anonymized Internet Traces Dataset (2019)*. https://www.caida.org/catalog/datasets/passive_dataset. Accessed: 2025-07-30. 2019.

[CCFC04] M. Charikar, K. Chen, and M. Farach-Colton. "Finding frequent items in data streams." In: *Theoretical Computer Science* 312.1 (2004). Automata, Languages and Programming, pp. 3–15. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(03)00400-6.

[CH08] G. Cormode and M. Hadjieleftheriou. "Finding frequent items in data streams." In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), 1530–1541. ISSN: 2150-8097. DOI: 10.14778/1454159.1454225.

[CM05] G. Cormode and S. Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications." In: *Journal of Algorithms* 55.1 (2005), pp. 58–75. ISSN: 0196-6774. DOI: https://doi.org/10.1016/j.jalgor.2003.12.001.

[HN+24] M. Hooshangi-Naghani, N. Rezaei, M. Dolati, A. Khonsari, and S. H. Rastegar. "IoT Threat Mitigation: Machine Learning-Assisted Heavy Hitter Detection in P4-Enabled Networks." In: *2024 5th CPSSI International Symposium on Cyber-Physical Systems (Applications and Theory) (CPSAT)*. 2024, pp. 1–8. DOI: 10.1109/CPSAT64082.2024.10745449.

[Hsu+19] C.-Y. Hsu, P. Indyk, D. Katabi, and A. Vakilian. "Learning-Based Frequency Estimation Algorithms." In: *International Conference on Learning Representations*. 2019.

[LRA22] O. Lavi-Rotbain and I. Arnon. "The learnability consequences of Zipfian distributions in language." In: *Cognition* 223 (2022), p. 105038. ISSN: 0010-0277. DOI: https://doi.org/10.1016/j.cognition.2022.105038.

[MAEA05a] A. Metwally, D. Agrawal, and A. El Abbadi. "Efficient Computation of Frequent and Top-k Elements in Data Streams." In: *Database Theory - ICDT 2005*. Ed. by T. Eiter and L. Libkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 398–412. ISBN: 978-3-540-30570-5.

[MAEA05b] A. Metwally, D. Agrawal, and A. El Abbadi. "Efficient Computation of Frequent and Top-k Elements in Data Streams." In: *Database Theory - ICDT 2005*. Ed. by T. Eiter and L. Libkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 398–412. ISBN: 978-3-540-30570-5.

[MG82] J. Misra and D. Gries. "Finding repeated elements." In: *Science of Computer Programming* 2.2 (1982), pp. 143–152. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(82)90012-0.

[MV22] M. Mitzenmacher and S. Vassilvitskii. "Algorithms with predictions." In: *Commun. ACM* 65.7 (June 2022), 33–35. ISSN: 0001-0782. DOI: 10.1145/3528087.

[New05] M. Newman. "Power laws, Pareto distributions and Zipf's law." In: *Contemporary Physics* 46.5 (2005), pp. 323–351. DOI: 10.1080/00107510500052444. eprint: https://doi.org/10.1080/00107510500052444.

[PCT06] G. Pass, A. Chowdhury, and C. Torgeson. "A picture of search." In: *Proceedings of the 1st International Conference on Scalable Information Systems*. InfoScale '06. Hong Kong: Association for Computing Machinery, 2006, 1–es. ISBN: 1595934286. DOI: 10.1145/1146847.1146848.

[RKA16] P. Roy, A. Khan, and G. Alonso. "Augmented Sketch: Faster and More Accurate Stream Processing." In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, 1449–1463. ISBN: 9781450335317. DOI: 10.1145/2882903.2882948.

[SM24] R. Shahout and M. Mitzenmacher. " Learning-Based Heavy Hitters and Flow Frequency Estimation in Streams." In: *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2024, pp. 1–13. DOI: 10.1109/ICNP61940.2024.10858542.

[YNS16] L. Yang, B. Ng, and W. K. G. Seah. "Heavy Hitter Detection and Identification in Software Defined Networking." In: *2016 25th International Conference on Computer Communication and Networks (ICCCN)*. 2016, pp. 1–10. DOI: 10.1109/ICCCN.2016.7568527.

[Zha+23] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Re, C. Barrett, Z. Wang, and B. Chen. "H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models." In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023.